Time-predictable Stack Caching

Abbaspourseyedi, Sahar

Publication date:
2016

Document Version
Publisher's PDF, also known as Version of record

Time-predictable Stack Caching

Sahar Abbaspourseyedi

Kongens Lyngby 2015
IMM-PhD-2015-385

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Matematiktorvet, building 303B,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3351
compute@compute.dtu.dk
www.compute.dtu.dk IMM-PhD-2015-385

Summary (English)

Embedded systems are computing systems for controlling and interacting with physical environments. Embedded systems with special timing constraints where the system needs to meet deadlines are referred to as real-time systems. In hard real-time systems, missing a deadline causes the system to fail completely. Thus, in systems with hard deadlines the worst-case execution time (WCET) of the real-time software running on them needs to be bounded.

Modern architectures use features such as pipelining and caches for improving the average performance. These features, however, make the WCET analysis more complicated and less imprecise. Time-predictable computer architectures provide solutions to this problem. As accesses to the data in caches are one source of timing unpredictability, devising methods for improving the time-predictability of caches are important. Stack data, with statically analyzable addresses, provides an opportunity to predict and tighten the WCET of accesses to data in caches.

In this thesis, we introduce the time-predictable stack cache design and implementation within a time-predictable processor. We introduce several optimizations to our design for tightening the WCET while keeping the time-predictability of the design intact. Moreover, we provide a solution for reducing the cost of context switching in a system using the stack cache. In design of these caches, we use custom hardware and compiler support for delivering time-predictable stack data accesses. Furthermore, for systems where compiler support or hardware changes are not practical, we propose and explore two different alternatives based on only software and only hardware support.

Summary (Danish)

Indlejrede systemer er computer systemer der kontrollere og interagere med den fysiske verden. Indlejrede systemer med specielle tidsrestriktioner, på at møde tidsfrister, bliver kaldt tidstro systemer. I hårde tidstro systemer, kan det at systemet ikke møder en tidsfrist have katastrofale følger. Derfor skal den værst mulige køre tid (WCET) af den tidstro software i systemer med hårde tidsfrister være begrænset.

Moderne arkitekture bruger funktioner så som, pipelining og cache for at forbedre den gennemsnitlige ydeevne. Disse funktioner gør i midlertidigt analysen af WCET mere kompliceret og mindre præcis. Tidsforudsigelige computer arkitekture søger at løse dette problem. Da det at tilgå data i cachene er en kilde til tidsuforudsigelighed, er det vigtigt at udtænke en metode til at forbedre forudsigeligheden af cachene. Stack data, med statisk analyserbare adresser, giver en mulighed for at forudsige og stramme WCET af at tilgå data i cachene.

I denne afhandling introducere vi designet af den tidsforudsigelige stack cache samt implementationen i en tidsforudsigelig processor. Vi introducere flere optimeringer af vores design for at stramme WCET, alt mens forudsigeligheden bevares intakt. Desuden giver vi en løsning der reducere prisen af et kontekst skifte i et system der bruger stack cachen. Som en del af at designe disse cache, bruger vi special lavet hardware og compiler support til at gøre adgang til den tidsforudsigelige stack cache. Endvidere forslår vi to alternativer til systemer hvor compiler support eller hardware ændringer ikke er praktisk mulige. Disse to alternative forslag basere sig på software support eller hardware support.

Preface

This thesis was prepared at the Department of Applied Mathematics and Computer Science at the Technical University of Denmark in partial fulfillment of the requirements for acquiring the Ph.D degree in Computer Science.

The work of this thesis was partially funded by the European Union's 7th Framework Programme project Time-predictable Multi-Core Architecture for Embedded Sys- tems (T-CREST) under grant agreement no. 288008. The results of this thesis are part of the T-CREST platform and available under the BSD open source license.

The thesis explores the topic of time-predictability regarding stack data accesses. We provide design and implementation of several time-predictable stack caches and report on their performance in the context of time-predictable Patmos processor.

The thesis does not contain any material that has been accepted for the award of any other degree or diploma in my name, in any university or other institution and, to the best of my knowledge does not contain any material previously published by another person, except where due reference is made in the text of the thesis.

Lyngby, 5-9-2015

Sahar Abbaspourseyedi

Acknowledgements

I would like to thank my supervisor, Martin Schoeberl, for giving me the opportunity to study at DTU as a PhD student and instructing me during this period.

Special thanks to Florian Brandner, for supporting me, specially during the final months of my studies. Without his help this work would have not been possible.

I would like to thank the fellow PhD students and colleagues at DTU and the T-CREST team, specially Laura.

Letizia, I cannot thank you enough for all the long long talks to convince me that I can do this and helping me see the end line.

Karin, thank you for helping with all the official matters and all the candies and smiles.

My brother, Mohsen, I am grateful that you exist in my life. Without your support I would have not been here.

My other brother, Mehdi, I love you, thank you for supporting me when I was a child and even if you had to struggle through a lot.

My parents, Shahrzad and Bakhtiar, I love you. Thank you for letting me choose my own path and always encouraging me by all means.

Finally, Fredrik, you brought back hope to my life. I am thankful for your

support and love during the past 3 years.

–Sahar Abbaspour, 15-02-2016

Contents

Chapter 1

Introduction

Embedded systems are hidden computing systems for controlling and interacting with physical environments. Application domain of the embedded systems can be very wide. Embedded systems can be found in nodes of a distributed sensor network, avionics, multimedia or even household applications, etc. Some embedded systems have special timing constraints, i.e. tasks running on the systems must meet deadlines otherwise the systems might fail. These systems are referred to as real-time systems. In the real-time systems, meeting the deadlines is as important as the system producing correct results. Real-time systems can be divided to three categories: 1) hard real-time systems, where missing any deadline causes a system failure, 2) firm real-time systems denote such systems for which a result produced after the deadline is useless and discarded, and 3) soft real-time systems can use the result that the tasks produce after their deadlines, resulting in lower performance for the systems.

Systems with hard deadlines have special timing requirements where the worst-case execution time (WCET) of the real-time software running on them needs to be bounded. The WCET estimation should not underestimate the real execution time. Moreover, it should be as tight as possible. For example, the air bag system in a car should respond within a certain time to minimize the risk of injury for the passengers [71].

Today's computer architectures provide high performances for the average case

scenarios. However, in hard real-time systems, determining and bounding the WCET of the system is even more important than the high average performance.

Modern architectures provide features for improving the average performance, features such as pipelining, caches, and branch predictors. On the other hand, these features cause an adverse effect, making the WCET analysis more complicated and less imprecise. Time-predictable computer architectures provide solutions to this problem by alternative designs. These designs aim for an easier and more precise timing analysis of the systems.

Memory sub-systems are of great importance for the performance of systems. More importantly, due to their effect on timing of the systems, they are crucial for time-predictability as well. Therefore, memory and caches and their analysis have been topics of a wide range of researches [68, 100].

Caches exploit the temporal and spatial locality of accesses to the items in the memory. To get a precise WCET bound, cache analysis needs to statically predicts hits and misses [26]. For this, the WCET analysis of the data caches requires precise knowledge about the addresses of accesses and takes the replacement policy into account [100] . With a known address of a load or store instruction, it is easy to determine whether the access is a hit or miss. However, statically unknown addresses of the heap allocated data in the data caches, specifically pose a bottleneck for determining the WCET. Thus, devising alternative methods of the cache design to tighten the WCET and simplify the analysis is crucial.

Distinguishing memory accesses of predictable and unpredictable data structures is proposed to improve the WCET of caches [53]. One important observation regarding the data caches is that we can statically determine the addresses of the stack allocated data. For the heap allocated data, on the other hand, we can only know the addresses at runtime (i.e, they are not statically predictable). Thus, separating the stack data to a dedicated cache can improve the time-predictability of the system. Another benefit from a separate cache for stack data is that the number of accesses to the data cache decreases and hence less number of cycles is spent on transferring between the slow main memory and the data cache. This is more important when using a write through cache that on any store to the cache requires a write to the main memory as well. Thus, a dedicated cache for the stack data can help reducing the WCET and improving the analysis precision.

The focus of this thesis is to thus to introduce the design and implementation of a novel and efficient time-predictable cache architecture for the stack data, in the context of the time-predictable Patmos processor. The challenge for designing such a cache is that the time-predictability of the design can not be

compromised for efficiency.

The main goals of this thesis are thus, (i) design and integration of the stack cache to the Patmos processor, and (ii) improving its average-case performance while keeping the time-predictability, i.e. the main feature of this design, intact. The analysis of WCET of the stack cache and the effect of the stack cache on improving the WCET of the processor are thus out of the scope of this work. However, the stack cache design is simple and provides means for easy analysis of its WCET behavior [41].

The stack cache is a window to the main memory and we implemented it using two processor-internal registers and three instructions designed to manipulate the stack cache. All of the loads and stores accessing the stack cache are hits and therefore, we guarantee a single cycle access time for them. We present the implementation of our design on Patmos [87] time-predictable processor. Furthermore, we report on our experiments results from implementing the software and hardware extensions. Moreover, we provide techniques for performance improvements of the proposed stack cache while preserving the time-predictability characteristic with minimal overheads. In addition, we propose an extension of the stack cache that allows to (partially) hide the overhead of saving and restoring the context switching. Finally, we look at the stack cache from two different angles: a software managed and a hardware managed stack cache. The software managed stack cache is a design for systems where changing the hardware is not an option. On the other hand, the hardware managed stack cache suggests using the stack cache for standard Microprocessor without Interlocked Pipeline Stages (MIPS) style processors where no compiler support or instruction set architecture (ISA) changes are needed. In more detail, our contributions are:

- *Time predictable stack cache:* We propose a novel cache design for the stack data. The stack cache is integrated to the time-predictable T-CREST platform. We provide the open-source implementation details and evaluate our design integrated to the time-predictable Patmos Processor.

- *Lazy spilling stack cache:* We propose and describe a method to improve the performance of the stack cache by reducing the unnecessary transfers between the stack cache and the main memory by introducing minimal changes to the original design while keeping the time-predictability as the main characteristic of the stack cache. We explain the details of the implementation and report on the performance improvement.

- *Block-aligned stack cache:* We propose and describe another method to improve the performance of the stack cache concerning the requirements of the aligned address of the transfers to/from the main memory's controller.

We integrate our changes with the Patmos processor and report on the speedup using this method.

- *Virtual stack caching:* We propose and describe a hardware extension to *virtualize* several stack caches in a shared memory space, which allows us to quickly switch between these virtual caches. The preemption overhead can partially be hidden through this extension, which is profitable for the Worst-Case Response Time (WCRT) of the preempting task. This furthermore opens promising opportunities to save/restore virtual caches of preempted tasks during the execution of the other tasks.
- *Software managed stack cache:* We describe a time predictable stack cache implemented in software and evaluate its utilization of the scratch-pad memory (SPM) dynamically. We integrate our design to the T-CREST [83] platform and evaluate the utilization of the SPM.
- *Hardware managed stack cache:* We describe design and implementation of a stack cache for stack allocated data for MIPS style processors. The differentiating factor of this design is that it requires no compiler support. We integrate the design with the Patmos processor as it has a similar architecture to MIPS processors and report on the performance gains.

This thesis is organized as follows: Chapter 2 provides the necessary background and related work on the time-predictable architectures and specifically time-predictability in caches and systems with shared caches. Chapter 3 describes the details of the design and implementation of the time-predictable stack cache and embedding it to the Patmos processor. Chapter 4 provides two methods to improve the performance of the time-predictable stack cache and their implementation details and a method to reduce the cost of context switching. This chapter is divided in three sections: (1) lazy spilling stack cache, (2) block-aligned stack cache, and (3) virtual stack caching. Chapter 5 presents the software and hardware managed stack caches. Firstly, we look into utilization of the the SPM as a software managed stack cache. Secondly, we discuss different characteristics of the hardware managed stack cache and the requirements to integrate it with the Patmos processor. Chapter 6 presents the evaluation of our proposed designs and ideas, integrated with the Patmos processor and the T-CREST platform. We present both the hardware overheads and average performance measurements using the cycle accurate simulations. We conclude this work in Chapter 7 with insights to future works.

Publications

- Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raf-

faele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015

- S. Abbaspour, F. Brandner, and M. Schoeberl. A time-predictable stack cache. In *Proc. of the Workshop on Software Technologies for Embedded and Ubiquitous Systems*. 2013

- S.Abbaspour, A. Jordan, and F. Brandner. Lazy spilling for a time-predictable stack cache: Implementation and analysis. In *Proc. of the International Workshop on Worst-Case Execution Time Analysis*, volume 39 of *OASICS*, pages 83–92. Schloss Dagstuhl, 2014

- S.Abbaspour and F. Brandner. Alignment of memory transfers of a time-predictable stack cache. In *Proc. of the 8th Junior Researcher Workshop on Real-Time Computing*, 2014

- Sahar Abbaspour, Florian Brandner, Amine Naji, and Jan Mathieu. Efficient context switching for the stack cache: Implementation and analysis. In *to appear in RTNS 2015: 23rd International Conference on Real-Time Networks and Systems*, Lille, France, November 4-6, 2015

Technical Reports

- D2.3 in T-CREST project - Hardware Implementation of Patmos. `http://www.t-crest.org/page/results`, 2014.

Chapter 2

Background and Related Work

As we mentioned in the first chapter, for the hard real-time systems, aside from the average performance, meeting the deadlines and a precise WCET estimation is a requirement. This however, is only possible if the hardware architecture of the system is time-predictable (i.e. the analysis of the architecture for obtaining precise WCET is feasible).

It is important to note that the high performance architectures might not result in better WCET estimations. This is due to the complexity of the architecture, making the analysis of the system more complex and pessimistic. Moreover, the variation in the execution time in architectures with more complicated mechanisms is higher and thus the observed execution time might not reflect the WCET accurately.

In this chapter we provide basic definitions and the necessary background on the timing analysis, time-predictable architectures and specifically time-predictable data and stack caching.

This chapter is organized as follows: in section 2.1 we briefly introduce the timing analysis and different methods of timing analysis. In section 2.2 we review several state of the art time-predictable architectures and explore their characteristics. In 2.2.6.1 we present a brief description of the time-predictable

Patmos Processor which is the embedded platform used for our time-predictable stack caching development. In section 2.3 we explain the effects of caches in time-predictability and WCET. Section 2.4 looks into the methods to improve time-predictability in the data caches. We specifically look into the effect of the data caches on the WCET of systems with shared caches. Splitting different data accesses to different caches is discussed in section 2.5. We further discuss the time-predictability issue in the scratchpad memories in 2.6. In section 2.7 we look into some of the architectures to support context switching. Finally, we summarize this chapter in section 2.8.

2.1 Timing analysis

As we mentioned, real-time systems run tasks with special timing requirements. In hard real-time systems, however, the tasks *must* meet their deadline. Therefore, we need analysis techniques to acquire precise timings for these systems. In the following, we look at different methods and aspects of timing analysis.

Initial state of the hardware usually affects the execution time of the system. For example, the contents of the data caches can impact the time to load the data from the data cache [71]. The WCET of a system depends on the different execution paths, generated by conditional branches as well. However, large numbers of inputs that have wide ranges, generate many possible execution paths, making it impossible to observe all the execution paths and the time they take. Thus, to have a bound on the WCET of the system, *timing analysis* methods are used [71].

Finding the WCET bound of a program includes three steps: (1) determining the possible execution paths or *flow analysis*, (2) calculating the execution time of the possible paths or *low level analysis*, and (3) choosing the longest possible execution path based on the results of steps (1) and (2).

Two different approaches can estimate the WCET of possible paths: (1) real execution of the program on real hardware or on a cycle-level simulator (measurement analysis), and (2) *static analysis*.

In the following we explain the three steps required for the static analysis. It should be noted that all these steps are required for measurement based analysis as well. However, the measurement analysis requires exploration of all the possible paths in the program and thus, is a costly method. Hence, we focus on the static analysis in this work. Further reading on the measurement analysis can be found in [71].

2.1.1 Flow analysis

For flow analysis, building a representation of the program is necessary. There are different methods for this. One method is to build a syntax tree of the source code. However, relating nodes of the tree to sequence of instructions might be difficult due to compiler optimizations. Therefore, usually, a control flow graph (CFG) is built using the executable code [15]. When the program does not include indirect branches with their target known at runtime, building the CFG is simple [16]. In [13], a solution to the indirect branch problem is proposed based on the decoding of branches' conditions and value analysis.

Determining values of loop bounds or frequency of conditional branches, the so called flow facts, is another part of the flow analysis. Determining the flow facts can be done by means of user annotations [45] or automated [19]. The automated analysis can be performed at the source code level or binary level. The binary level is more useful when the program uses library functions and there is no access to the source code [71].

2.1.2 Low level analysis

The low level analysis tries to determine the execution time of the basic blocks of the code based on the target architecture running the programs. Therefore, details of the hardware architecture such as pipeline, caches and branch predictor should be known. When enough details of the architecture is not available, description languages are used to build a realistic model of the hardware [75].

Since the execution of the previous blocks changes the state of the hardware (e.g. pipeline and memory system), the execution time of a basic block is usually dependent on the execution history. Usually, techniques such as abstract interpretation [17] are used to consider all of the possible paths. Static analysis methods simulate the execution of the basic blocks with all the possible initial states (or parametrized initial states [76]) to determine the WCET of each basic block.

2.1.3 Computing WCET

After the first two steps (2.1.1 and 2.1.2), we can compute the WCET of the whole program. For the WCET computation, we need to choose a program representation. There are two different types of program representations: *syntax*

tree and *CFGs* [71]. In the syntax tree representation, the nodes represent the control flow statements such as loops and conditionals and the leaves are the basic blocks. In the CFGs however, the nodes represent the basic blocks and the edges represent the control flows. The WCET is then computed according to the following:

- Tree-based approaches: a weight is assigned to each basic block according to its WCET. The weights are added and propagated from the leaves to the root of the tree [63]. Although this method is simple, for cases where the compiler optimizations change the code structure, it may not be useful.
- The CFG methods: this method uses the weighted sum of the basic blocks for calculating the global WCET. The weights are based on the execution count of the basic blocks. Then, considering flow facts as constraints, using solving tools such as lp_solve, the maximum for the weighted sum is calculated (i.e. the WCET) [49].

A number of commercial and academic research tools have been developed for the WCET analysis, such as *aiT* [36], *Otwata* [9] and *Sweet* [50].

Besides the mentioned methods, there are several other techniques to compute the WCET. Model checking and probabilistic methods are some examples of alternative solutions. However, these methods may not be applicable in systems with complex hardware architecture [71].

In the next section, we introduce some of the state of the art time-predictable architectures that are mainly designed for predictable WCET with reasonable average case performance.

2.2 Time-predictable architectures

In this section we discuss about some of the state of the art time-predictable architectures. However, before talking about that we introduce the notion of predictability.

2.2.1 Predictability definition

Thiele et al. [89] define the predictability based on the notion of the execution time, i.e. the time interval between a set of specific events. Thus, the maximum

and minimum of these timing interval define the worst and best case execution times under all possible scenarios of system and environment states. For these best and worst cases, lower and upper bounds can be computed by different methods such as analysis or simulation. They introduce the difference between these upper and lower bounds as a mean to determine the time-predictability. Thiele et al. classify the reasons for varying the timing interval to: interference and limited analyzability. Limited analyzability refers to the cases such as complex cache replacement strategies which may not result in close bounds to the execution time.

Kirner et al. [46] present different interpretations of the predictability when it is statement about the phenomenon execution time. One interpretation is that predictability describers the deterministic behavior of the hardware and software. The second interpretation is that for a given set on inputs and outputs there are minimum and maximum execution times and the interval between these two is where all possible execution times of tasks can be found. Finally, the predictability can describe the development of a task's execution time over time. For example for a periodic task there is a repeated pattern. They define time-predictability as the ability of calculation the durations for actions on the concrete system. Specifically they define predictability as the multiplication of analyzability and stability This ability refers to tractability rather than decidability. Therefore, a precise calculation of system's timing behavior must be available. Based on this, they propose the timing barrier concept. Which describes mechanisms for limiting the propagation of timing effects in architectures.

Grund et al. [34] state that there are two aspects to predictability. First, the property to be predicted and second, the sources of uncertainty. Finally, we should be able to compare the predictability of different systems and thus, the last aspect of predictability is a quality measure on the predictions. They explain that these aspects can be further refined, for example, considering different sources of uncertainty separately. They present a the quotient of the best-case execution time (BCET) over WCET as a reasonable quality measure for time-predictability; the smaller the difference the better the predictability.

In this thesis, time-predictability is the the ability to derive a WCET bound statically. Moreover, it is important to be able to tighten this value.

We explained the notion of predictability. However, it is important to note that the conventional architectures target the improvement of the average case execution time at the expense of variation in the execution time which leads to non-predictable timing behaviors. In the rest of this section, we explore some architectures that are specifically built to be time-predictable.

2.2.2 Precision Timed Machines (PRET)

Temporal properties for the instructions are often not specified in the modern ISAs. Therefore, implementations are not obligated to exhibit predictable and repeatable timing behaviors. Hence, bringing temporal properties to the ISA allows reasoning about the timing of the programs independent of the platform [51].

The simplest way to extend the ISA with the timing properties is to associate constant execution time to the instructions. The drawback of this method is that it prevents performance improvements at the micro-architectural level. Adding instructions to the ISA to control the timing behaviors of programs is proposed in [39], where a deadline instruction allows the programmer to specify the minimum execution time of the code block.

This idea can be further expanded to introduce a set of new assembly instructions to control not only the minimum execution time, but also the cases where the execution time exceeds a deadline. Defining timing instructions allows unifying the semantics of time across all the programs and requires any implementation to conform to the timing specifications in the software. The ARM ISA sets an instruction encoding space for co-processor extensions. Therefore, it can be used as a basis for implementing the timing instructions. In [51] three timing instruction are added to the ARM ISA are as follows:

- An instruction to obtain the time on the platform clock. It loads the 64 bits time stamps of the current platform to the general purpose registers. Although this can be implemented by memory mapping, there is no guarantee that the loaded value from main memory contains a consistent time stamp value from the same point in time.
- An instruction to take a time stamp and compare it to the current platform time to determine if delays are needed. As an example, a minimum execution time can be added to the time collected using the get_time to compute a deadline time. At the time of executing the *delay_Until instruction, if the platform time is less than the deadline time, the program will be delayed. This instruction can be used as a way to synchronize programs with the external resources.*
- *An instruction to specify a maximum execution time for code blocks. Decoding this instruction generates a timeout value. When the platform time exceeds this timeout value, an exception is thrown in the hardware. Moreover, another instruction is added to deactivate the timeout value in the hardware before an exception occurs. Multiple deadlines can be managed*

in the software using data structures that are keeping an ordered list of deadlines.

None of these instructions enforces execution behavior. Thus, architecture implementation and performance is not affected by this extension. The desired execution time of the programs or functions can be obtained from a higher level model or specification of the application.

These instruction extensions allow the programs to specify execution times in software that are necessary for the correct execution of the program, e.g. a minimum execution time for the code blocks, and more importantly detecting and handling the missed deadlines from code blocks with variable execution times

Different timing behaviors can be constructed for a code block (task) using these instructions:

- Scenario A If a task modifies the external I/O which can not be left in unknown states, then the task should be completed before further action (e.g. executing a miss handler in case the deadline is not met).
- Scenario B If the missed deadline should be handled immediately, the *exception_ on_ expire and* deactivate_exception instructions are used.
- Scenario C If a minimum execution time should be elapsed, the *delay_ Until instruction can be used. This instruction should be added after the* exception_on_expire instruction to check the task is not violating the deadline time. Furthermore, it should be added after the *deactivate_ exception instruction. Otherwise, the exception would always be thrown.*

To improve the average case performance of pipelines, multithreaded architectures can be used to overlap the execution of different instruction from different hardware threads. Hardware threads have their own copy of the processor state (e.g. register file and program counter) and the rest of the pipeline remains the same as the classic five stage RISC. When a pipeline hazard happens, during the execution of one thread, another thread can be fetched and executed. There are two thread scheduling policies. In a coarse-grain policy, threads continue execution until a context switch is triggered. Fine-grain policy switches context more frequently, for example on every clock cycle. Multithreaded architecture bring up problems regarding static execution time analysis. For example if a hardware component is shared between the threads, its internal state can be modified during the context switch and affect the execution time of the individual instructions. Therefore, several architectural adjustments should be

considered to achieve a time-predictable architecture. [51] We can consider a thread-interleaved pipeline with fine-grain thread scheduling, where on every cycle, a different thread is fetched. With a round robin scheduling policy, after 4 clock cycles, each pipeline stage is occupied by a different hardware thread. In order for this architecture to be time-predictable, threads must be temporally isolated. Several features should be refined for temporal isolation of threads:

- Control Hazards: The minimum number of the threads to eliminate the control hazards is dependent on the number of pipeline stages. To keep the safe side, the same number of hardware threads as pipeline stages remove control hazards. This is due to the fact that at any point of time, each pipeline stage is executing a different thread, and therefore there is no explicit dependency between different instruction in the pipeline.

- Data Hazards: The same reason for no control hazards in thread-interleaved pipelines, applies to the data hazards as well. However, long latency operations (e.g. memory operations) still can cause data hazards if there is no more thread to fill the pipeline with. This can be eliminated by inserting stalls to remove dependencies between the pipeline stages and therefore the threads will still be temporally isolated. Since there is no data dependency in this architecture, the forwarding and data hazard detection hardware units can be eliminated.

- Structural hazards: In multithreaded pipelines, if a thread accesses a multi-cycle hardware unit, another thread should wait to access the same unit. Therefore, multi-cycle hardware units should be pipelined to be accessible within one clock cycle. If it is not possible to pipeline the hardware unit, other mechanisms should be devised. For example, in the case of a memory unit that is non-pipeline-able, a time division multiplex access (TDMA) schedule can remove timing interferences.

Moreover, shared cache among several threads in a multi-threaded architecture, can cause a problem by influencing the execution history and changing the state of the cache. In PRET, the scratchpad memory of each thread is independent and thus the threads can be analyzed independently.

The stack cache presented in this thesis, is processor local. However, on a context switch the entire stack cache should be saved to the main memory and on restored when the preempted thread starts running again. In Chapter 4 we present a method for reducing the cost of the context switching for the stack cache.

2.2.3 Java Optimized Processor (JOP)

JOP is a small processor core for embedded Java systems [78] with time-predictable execution of Java programs. JOP is a RISC processor compared to the Java virtual machine (JVM) which is a CISC architecture. However, like the JVM, JOP is a stack machine. JOP works with its special instruction set called microcode. It translates the Java bytecodes to microcode instructions. The JOP pipeline has three stages, fetch, decode and execute. It has one extra stage in the front of the fetch stage, and translates the Java bytecodes to addresses in microcode. The fetch stage fetches the microcodes from the internal memory. The decode stage generates addresses to access the stack RAM as well as decoding the microcodes. The last stage performs ALU and memory operations (including stack accesses) using the two topmost elements of the stack. Short branch delays are straight results of the short pipeline, therefore hard to analyze branch prediction can be avoided. The Java microcodes in the JOP are native to the processor, i.e. they can have direct access to memory and I/O devices without the help of any other language. Moreover, using microcodes to implement the Java bytecodes simplifies the calculation of the WCET for bytecodes.

JOP supports two time-predictable caches: a stack cache and a method cache. A simple dual-ported memory with one read and one write port inside the JOP pipeline is used for the stack management. With the simple implementation of the stack memory, implementation of a large stack is possible, allowing holding the variables of nested functions.

2.2.4 The multi-issue CAR-Core processor

The CAR-Core processor [92] has a similar pipeline to the Infineon's TriCore 1 microcontroller [1]. This processor can execute up to four threads in parallel. The processor includes two pipelines with eight register sets (one register for each of the pipelines and one for each hardware thread slot), and the scheduler. One of the pipelines is used for data calculation and the other pipeline is for address calculation. For a predictable behavior, caches and speculation are avoided and threads can not influence the run-time behavior of each other. TriCore instructions such as call and return are complex and occupy the address calculation pipeline for several cycles. Therefore, these instructions can affect the time predictability of the pipeline. To solve this problem, CAR-Core implements these complex instructions as sequences of microinstructions that are interruptible. Therefore, each of the threads can interrupt execution of any other thread's program and execute its own microinstructions.

A thread can lock the load/store interface with long memory latencies. Therefore, the CAR-Core implements split-phase loads, i.e. the address calculation and the writing of the data in a load operation are done separately. The received data from the memory is stored in a load buffer and is written to the register file later.

In the following section we introduce two time-predictable multi-core architectures.

2.2.5 A hard real-time capable multi-core SMT processor

In a time-predictable mutli-core architecture, timing composability can be provided by making WCET estimation of each task independent from other tasks in the task set.

In MERASA architecture [93], simultaneous multithreaded (SMT) cores run a single hard real-time (HRT) and up to three non-hard real-time (NHRT) tasks in parallel. Full isolation between tasks within a core provides time bounded interaction of HRT tasks. This architecture provides the following characteristics in the core and multi-core levels.

Cores are in-order superscalar pipelines. Each core has an integer and an address pipeline. The fetch stage, privileges the task with the highest priority (HRT task). Then the Real-Time Issue (RTI) stage receives the fetched instruction and insert it to the instruction window of the appropriate thread slot. Furthermore, it manages multi-cycle instructions. The multi-cycle instructions are interruptible in order for the HRT task to access the resources as soon as needed.

To handle memory operations, a split phase load technique is applied. Once a load instruction is recognized, the first part of the instruction (memory address calculation) is forwarded to intra-core real-time arbiter (ICRTA). Later when the data is available to the ICRTA, the second part of the instruction that writes the data to the register set is issued. The second part is omitted for store instructions.

First level instruction and data caches provide separate accesses for HRT and NHRT tasks. Dynamic instruction scratchpad (D-ISP) loads the complete function code of HRT tasks. Loading the function to the D- ISP stalls the processor. The D-ISP consists of a controller, function memory, mapping table, lookup table and a content stack. The D-ISP controller itself contains fetch control, content management, and the context register. The fetch control delivers the

instructions to the pipeline. It uses the context register to look up the active function in the context register.

The content management stores the mapping information of the function into the context register. Looking up the entry on a call is simple because the address is directly generated by the target of the call instruction. On a return, the address is unknown, therefore, a function address stack is used to keep track of return addresses. If a function is not available it is requested from the ICRTA, the ICRTA brings the function from higher memory levels. To load a function to D-ISP function size is needed. This information is provided by the compilation and an instrumentation tool hooks it to the beginning of every function with a special instruction. If the function is larger than the unused function memory, with a FIFO replacement policy the memory is over-written.

On multi-core level, the MERASA architecture uses an on-chip bus to obtain the requested blocks from higher memory levels. When two or more requests are sent to the bus, an arbitration policy is needed. Different bus arbiters are exploited for scheduling requests within each core and between different cores. Accesses to the bus can be due to (1) filling D-ISP, (2) data or instruction cache load miss, and (3) store for NHRT tasks. These requests from different cores are kept in separate queues for each core and therefore, the WCET only depends on the number of cores and not the total number of tasks from other cores.

The ICRTA, stores the request to the Dynamically Partitioned Cache, into different banks based on the target destination. Requests from different banks are handled based on FIFO policy for HRT tasks. For NHRT tasks, in order to increase the performance, out of order execution of different cache requests based on the First Ready First Served policy is applied. The inter-core bus arbiter (XCBA) requires that requests from HRT tasks are not delayed more than a given upper bound delay (UBD). When different HRT tasks trying to get access to the bus, a round robin policy is applied.

Therefore, the maximum UBD for an HRT task is defined by, $(N_{HRT} - 1) * L_{bus}$, where N_{HRT} is the number of HRT tasks (upper bounded by the number of cores) and L_{bus} is the delay incurred by the bus. The combination of delays from HRT and NHRT tasks determines the maximum delay, which is given by: $(N_{HRT} - 1) * L_{bus} + L_{bus} - 1$. Shared caches are source of interference as well. There are two sources of interference in memories in multi-core systems: (1) banks access interference, where two different memory request are addressed to the same bank. In XCBA this kind of conflict is avoided by delaying the second access to the same bank, and (2) storage Interferences, where useful data from a task is evicted by another task. This problem is addressed with partitioning the cache (Bankization). To find the UBD for each task, the processors are extended with a WCET computation mode. In this mode, each tasks is run

in isolation, and on each access to the shared resources, the access is delayed artificially by the UBD (either memory or bus delay) of the that access.

2.2.6 Time-predictable multi-core architecture (T-CREST)

T-CREST is a time-predictable multi-core platform [83]. T-CREST platform aims for two objectives: the fast worst-case and easy analysis. A memory tree [28] connects several Patmos processors to a real-time memory controller [30] for the shared, external SDRAM memory. A time-predictable network-on-chip [85] provides efficient core-to-core communication, while reducing the demands from the shared memory bandwidth.

The time-predictable core-to-core network-on-chip (NoC) [44], in the T-CREST platform is statically scheduled and time-division multiplexed (TDM). Each processor core has a local memory. The NoC exchanges data between processors using these local memories. The NoC supports asynchronous message passing using DMA-driven block-transfers. The TDM-based NoC choice avoids dynamic arbitration and virtual channel buffers, thus providing simple hardware implementation. Moreover, TDM provides straightforward timing analysis.

The Bluetree memory tree [28], is a set of 2 to 1 full-duplex multiplexers and allows the communication from a set of processing nodes to the main memory and does not allow communication between core processors (this is left to the NoC). The memory tree distributes memory arbitration across the routers instead of using a large monolithic arbiter close to the main memory. This allows a higher clock frequency.

The Patmos compiler [64] is an adaption of the LLVM compiler. The compiler is able to generate single path codes. Thus, eliminating the sources of uncertainty in the software. Moreover, the aiT tool [36] is extended to support the Patmos architecture. aiT takes binary executables and annotation file as inputs. To support the integration of the analysis tool, the compiler should pass information about machine code to the WCET analysis tool [11].

The Patmos [84] processor provide the core processor for the T-CREST [2] project. Patmos' design allows the tight WCET while considering the average performance as well. The Patmos processor has a dual-issue RISC pipeline. Instruction delays are visible through the ISA to simplify WCET analysis. Several special caches (data cache, stack cache, method cache) provide the cache splitting strategy to help improve the WCET bounds. For unpredictable data accessed, cache bypassing is used. Using CoreMark benchmark, Patmos is compared to other FPGA based processors such as the Aeroflex Gaisler LEON3,

Xilinx MicroBlaze, and Altera NIOS II processors. Results show that Patmos offers the same performance while it provides time-predictability. Since we use the Patmos processor to integrate and evaluate the ideas presented in later chapters in this thesis, here we explain the Patmos processor in more details.

2.2.6.1 Patmos processor

In the following we explain different features of the Patmos processor.

Patmos ISA Patmos has a dual-issue RISC alike load/store instruction set. Each instruction can use up to three register operands. There are eight predicate registers and logical operations like AND and OR, change the value of these registers. Compare instructions set predicates values. Instruction length can be either 32 or 64 bits. The first bit of the instruction indicates the length of the instruction. Predicating all instructions facilitates the single path code generation as well as reducing the number of conditional branches. The first instruction of an instruction bundle contains a bit to encode the length of the bundle (32 or 64 bits). Fetching two 32-bit words for the dual-issue pipeline allows using the second instruction as a constant and thus loading 32-bit constants in a single cycle. However, most of the ALU instructions work with 12-bit constant operands. Therefore, the second instruction slot can contain instructions instead of data.

Different data areas (e.g., stack and local data) can be accessed with typed load and store instructions. Allowing the cache type identification earlier in the pipeline.

Patmos Pipeline Patmos processor's pipeline has five stages. The basic functionality of the pipeline stages are similar to RISC processors:

- Fetch: fetches the instruction from the method cache.
- Decode: decode stage contains the 32 bit register file. In this stage source operands of the instruction are read from the register file. The two pipelines share the register file and support forwarding the values to the other pipeline. Since the pipeline is dual-issue, register file has four read ports and two write ports. Moreover, the decode stage generates the control signals for the next stages.

- Execute: execution stage reads the value of predicate registers and according to their values executes instructions. Moreover, the execution stage calculates the effective address for accessing different types of memories.
- Memory: in this stage reading/writing from/to different areas of the memory happens. Memory operations may stall the pipeline operation.
- Write-back (WB): writes the result to the destination register.

Besides the register file, Patmos contains several special registers: the S registers. Type of the instruction that uses these special registers defines where these registers are read or written to. Special instructions *mfs* and *mts* are used to read and write these registers. In the next chapters we introduce two of these special registers that are used in the stack cache design.

Local Memories Three caches (method, data, and stack cache) and two scratch-pad memories (SPM) for data and instructions in Patmos are on-chip and their sizes are configurable. The type information to access different memories is defined by the compiler (e.g. for the stack cache) or by the programmer (e.g. for a data SPM). The SPMs can be used in addition to caches or instead of them.

Method Cache Patmos' method cache stores entire functions. Patmos may load functions on a function call or return from a function. Instructions accessed in other times are guaranteed cache hits. The method cache design assumes that the cache is larger than any function in a program. At the source code level this assumption is not guaranteed. In case the size of the function is larger than the cache size, the *function splitter* in the compiler splits the functions into smaller functions fitting in the cache.

Data Cache Data cache in the Patmos processor is a direct-mapped or set-associative cache with write-through strategy and no allocation on a write. The write-through cache choice is to help the WCET analysis. For statically unknown load and store addresses, Patmos bypasses all the caches (Patmos compiler replaces the unpredictable accesses with the bypass instructions) Patmos also supports a small buffer to combine writes into bursts to improve the performance deflection that write-through policy causes.

We reviewed several time-predictable architectures. In these architectures, the memory hierarchy and caches, specifically, play a major role in increasing the

performance of the processor. As the sizes of the memory blocks increase, their access times increase as well, i.e. the fastest memory (i.e. cache) is the one nearest to the processor and the main memory is the slowest one. In the following sections we discuss caches and their effects on time-predictability in details. Before that, as most of the architectures we explained above use pipeline, in the following, we first briefly discuss the effect of the pipeline on the timing analysis.

2.2.7 Effect of pipeline on timing analysis

To analyse the pipeline, a pipeline model should be developed. This model can be a huge finite state machine because of the complexity of modern processors. Several components which are not affecting the timing model can be eliminated. Moreover, components of the pipeline are partitioned to units in which elements are closely related. These units communicate with signals which represent events like fetching an instruction. If an event is happening in the next clock cycle, it is delayed in this model. The analysis is done by collecting the set of pipeline states that occur during the execution of an instruction at any point. This gives an upper bound on WCET for that instruction [47]. Therefore, a methodology to analyze the WCET for a modern processor needs both the abstract interpretation of modules and pipeline modeling [76].

2.3 WCET analysis in caches

This section starts by explaining the basics of the caches, different types of caches and replacement policies. Then, we discuss the effect of various replacement policies on WCET of caches.

2.3.1 Caches in memory hierarchy

The growing gap between the processor speed and the memory speed is a drawback in the overall performance of systems [54]. Therefore, increasing the memory performance is essential for improving the performance of the whole system. Modern architectures provide caches for improving the average performance and covering this gap.

Caches are important parts of memory systems. A cache is a small but fast storage that resides between the processor and the main memory. Caches work

based on the locality principal to accesses the items in the memory, i.e. a program accesses a small part of the address space in a small window of time. There are two types of locality [61]:

- Temporal locality (time locality): when an address is accessed, the same address might be accessed soon again.
- Spatial locality (space locality): when an address is accessed, the addresses close to it might be accesses soon as well.

The first time an item is accessed from the main memory, it is copied to the cache as well. This is called a cache miss. The next time that this item is accessed again, it already resides in the cache and there is no need to transfer it from the slow higher level memory. This is called a cache hit. The simplest form of a cache is a direct-mapped cache. A direct-mapped cache can be seen as an indexed array of memory blocks or lines. Each line can be assigned to several memory addresses. To differentiate between these addresses a tag is assigned to each block. If a miss occurs, one of the lines in the cache may get replaced by the new line. This will cause the processor some penalty cycles. To reduce the rate of replacement of the blocks, a number of locations can be assigned to a single line. This is called set associative cache, offering a more flexible cache design. In the extreme form of a set associative cache, a line can be assigned to any memory address: the fully associative cache. To access the appropriate block in a set associative cache, the line address is divided to three parts [61]:

- Offset: the lower bits of the address, indicating the byte or word that is accessed.
- Index: the index part of the address to identify the number of the accessed set.
- Tag: part of the address that is stored on the cache. On each access to the cache all the tags of a set are compared to the tag part of the accessed address to detect a cache hit.

As we explained, several levels of caches can form a memory hierarchy where each higher level contains all the items of a lower level in it [61]. Therefore, whenever a block of memory is not available in the cache, a miss happens and the block is copied from a higher level (farther from processor) to the cache. For this replacement, several different polices exist [71]:

First in, first out (FIFO) or round robin (RR) a counter is assigned to each cache set to represent the most recent cache line that was written to after a cache miss. On a new miss, the cache replaces the next line's data and increments the counter. Once the counter reaches the last line in the cache, the counter is set to point to the first line again.

Pseudo round robin (P-RR) the P-RR strategy is similar to the FIFO replacement method. However, the P-RR method uses a single counter for each set. Therefore, cache miss in a set affects the contents of other cache sets.

Most recently used (MRU) the MRU strategy uses a bit mask (set to one when the line is accessed) for each of the cache lines to prevent replacement of the MRU cache line on a cache miss. The cache line with a 0 mask is evicted on a cache miss and its mask is set to 1. When there are more lines with the bit mask set to 0, one of them (depending on the time when their mask was inverted) is replaced.

Least recently used (LRU) this strategy keeps track of the order that the lines are used. When a line is accessed, the line moves to the beginning of the list and when the cache needs to evict a line, the last line on the list is replaced.

Pseudo-LRU (P-LRU) on a cache miss, the P-LRU strategy evicts one of the lines that are not recently used. A binary tree with the cache lines as leafs represents this strategy. Each of the nodes is associated with a bit indicating whether the left or right sub tree should be followed. On a hit, the bits on the way to the leaf associated with the cache line are set to the direction pointing to the opposite way of that leaf for next cache eviction.

Another important aspect of cache design is writing policy. The write policy determines when the data should be written to the main memory. Either the data is written to the main memory whenever it is written to the cache or it is only written when the data is evicted due to a cache miss. These methods are called write through and write back, respectively.

2.3.2 Static analysis of caches

For predicting the timing behavior of caches, we need to consider two issues [71]. First, we need to determine whether each cache access results in a hit or miss. Second, the access time on a cache hit or miss should be known. A common technique for categorizing cache accesses is introduced in [57]. Muller et al. propose three categories: 1) AlwaysHit, i.e. any access resulting in cache hit, 2) AlwaysMiss, i.e. any access to the specific data is always resulting in cache miss, and 3) NotClassified, i.e. the analysis cannot classify the access as one of the first two categories.

One major problem with the cache analysis is the address analysis. Accurate cache analysis needs to know about the addresses of the accesses (to determine whether they result in hit or miss). An unpredictable address leads to a cache miss prediction. Moreover, it is unclear whether it replaces other data in the cache, and if yes, which data. Thus, knowing the range of the addresses for a cache access can help reducing the overestimation of the cache miss access penalty. Thus, in cache analysis, we should consider the following problems:

- Determining the address of the memory access. For a data access, the address might be calculated dynamically at run time. Hence, the address is not available for the static analysis.

- Is the accessed address available in the cache, i.e. whether it is a hit or miss?

2.3.3 Effect of cache replacement policies on WCET

Unknown initial states and different sets of inputs are the factors effecting the cache analysis. Access to cache with unknown addresses, makes determining whether a cache access is a hit or miss, impossible. Assuming two miss penalties for a cache access with unknown address is pessimistic and results in imprecise WCET. Therefore, to obtain tight bounds on the execution time of a system, a precise cache analysis is crucial.

Each of the replacement methods influences the WCET of the system in a way. In the following we discuss the effect of different cache replacement policies on WCET [71].

LRU Replacement Policy In LRU replacement policy, the block which has been least recently used is replaced. To analyze the WCET in case of LRU caches, concrete cache states can be used in which each block has an age, and an update function changes the age of the block in every access to the cache. If the set of all concrete cache states are computed for each program point, a full cache analysis can be performed. In practice, memory consumption of such an analysis is prohibitive. Thus, *must* and *may* analyses are developed instead. The must analysis approximates an upper bound for concrete ages and finds all the blocks that must be in the cache at a special program point. The may analysis is dual to the must analysis and provides lower bounds for ages of the cache blocks. In other words, it defines all the blocks that are not in the cache. With the must and may analyses together, intervals can be computed for the cache states [33].

Pseudo-round-robin Replacement Policy As we explained, in pseudo-round-robin policy, replacement is controlled by a counter. The counter determines which line of the cache should be replaced. With this policy, some blocks remain in the cache forever and some are replaced frequently. Analyzing this cache should contain a model for the counter, but in case of unknown initial cache state, it is unknown what should happen to the counter. Therefore, the initial may set never changes. In case of the must analysis, it is not known which block is thrown out from the cache and it is possible to throw out all the blocks from the cache and the cache can contain only the missing block. Thus, there is only one memory block which is certainly in the cache. For example, if the cache has 4 lines, this analysis covers only 1/4 of the cache [99].

Pseudo-LRU Replacement Policy In a pseudo-LRU replacement policy, a number of bits are used to define the state of each cache line. Whenever a line is replaced, the bits indicating that line are negated. The behavior of the cache using this policy shows the lines are replaced with some strange order which results in uncertain analysis [33].

In all the cases discussed, it is assumed that the addresses of memory accesses are statically known. This is true for data access with absolute addressing. When the addresses of indirectly accessed memories are unknown at compile time, the set of possible target addresses should be reduced. This is done by the value analysis. In the value analysis, an interval of possible values is computed for each processor register. In some cases, this approximation detects some unfeasible path and thus the number of possible scenarios is reduced.

Since LRU caches have been shown to have the best predictability properties

of all set associative caches [89], we leave further discussions about the effect of other types of replacement policies on WCET to the interested reader.

2.3.4 Effect of pipeline on cache analysis

Cache analysis may depend on the pipeline. For example, prefetching instructions damages the instruction cache. This damage consist of the replacement of potentially useful instructions from the cache based on the conditional branch. Therefore, the maximal damage based on the branch behavior should be calculated. In a unified cache, this may result in replacement of data blocks by instructions. If the cache analysis is done before the pipeline analysis, the amount of this interference can not be determined precisely.

We discussed the effect of caches on the WCET analysis. In the following section we focus on WCET specifically in data caches. Usually the accesses to the memory in the processors are divided to two parts: the instruction and the data path. This is called Harvard architecture [27] Separating the data and instruction paths improves the performance and increases the timing predictability of the processor [71]. This is in contrast to the von Neumann architecture [18] that uses unified data and instruction path. Separate data and instruction memories (and hence caches), offer better time-predictability. Time-predictable instruction caches are out of the scope of this work and further discussion on instruction caches can be found on [79] and [102].

2.4 Improving the time-predictability in data caches

As we discussed, caches are important in the way that they influence the WCET of systems. Therefore, several methods have been proposed to improve the WCET of caches.

As we discussed, knowing the address of an access to the memory can help predicting if the access to that address causes a hit or miss. In data caches, the accessed addresses may be dynamically determined at runtime. In general, the data types in data cache can belong to one of the following categories [71]:

Stack data The stack data includes function's parameters and local variables and return addresses. It has been shown that the addresses of the stack data can be determined statically [67].

Pointers to data When pointers are to the statically allocated data, the addresses can be determined using the abstract interpretation. However, addresses of pointers to heap data are determined at the run-time and are thus not know statically [25]. Therefore, for time-predictability, this kind of data should be avoided.

Array elements accessed during execution of loops On each iteration of the loop, the addresses of element can change. Therefore, the load and stores instructions on each iteration of the loop can have different addresses. It has been shown that value analysis can determine these addresses [38].

Based on the above categories, to improve the time-predictability of caches, several methods have been proposed. In the following section we explore some of these methods.

2.4.1 Cache locking

Some cache lines or the entire cache is locked and can no longer be used. Meaning the content of locked blocks is fixed and cannot be evicted. Locking the cache entirely during the complete execution time of a task, if not all the required data fits into the cache (which will normally be the case), leads to a high-cache miss rate. As a result, the task gains predictability but, on the other hand, it will nearly be as slow as not using a cache at all. A compiler based technique for locking is introduced in [94] that has minimal performance loss. A lock/unlock statement is inserted to the source code and whenever the processor encounters the statement, it locks the data cache. The cache is unlocked when the processor encounters an unlock instruction.

2.4.2 Domain caches

Multiple data caches in a system can improve the time-predictability. Each cache can serve a different domain, e.g. the locking mechanism can switch between the caches, for locked and unlocked accesses. Another example of domain caches is a cache assigned to data used inside the loops. Downside of using multiple caches is cache consistency, e.g. the same memory address can be accessed outside as well as inside a loop. Therefore, the data inside the corresponding cache should be updated in all the caches in case of a write operation [71].

2.4.3 Dedicated caches

In contrast to domain caches, dedicated caches are more flexible. Dedicated caches separate the accesses depending on the type of the memory access. For example, accesses to statically assigned memory accesses use a different cache than the dynamically allocated arrays. Write-back techniques are not applied to dedicated caches [71].

In systems with shared caches, time-predictability should consider the interference between different parts of the systems using the shared cache as well. We look into this issue in the following.

2.4.4 Improving the time-predictability of shared caches

Cache interference in systems with shared caches can be a bottleneck in determining WCET as well. In a system with shared caches, a cache block from a task may be replaced by a block from another preemptive task. If the evicted block from the preempted task is used when the preemptive task is finished, an interference miss occurs. This is different from a cold miss occurring inside a task.

In the following we explain two different methods for improving the time-predictability in systems with shared caches.

2.4.4.1 Eliminating inter-process cache interference

In [66] a simulation-based approach is developed to identify if a cache miss is due to interference from other tasks. In this approach, each tasks owns its local cache, while a global cache is shared among all the tasks. Since the only way a cache line can be evicted from local caches is by each task itself, the local caches keep track of liveliness states of cache lines in essence. Therefore, four different types of access to caches can be observed:

- Global Hit, Local Hit: This signifies a normal cache hit.
- Global Miss, Local Miss: This corresponds to a miss that whether there were other tasks, it occurs. Therefore, it is a normal miss.
- Global Miss, Local Hit: This is considered as an interference miss.

- Global Hit, Local Miss: This is an impossible situation, since the global data is always a subset of what is in the local cache.

Using the SimpleScalar simulation infrastructure memory traces are generated for applications from the MediaBench and the MiBench benchmarks suits. Different cache configurations (4 and 8 way set associativity) has been examined. Applications with strong temporal or spatial locality suffer from interference significantly. On the other hand, in applications with high miss rate, the effect of interference is relatively small. Poor temporal locality and good spatial locality in streaming applications causes increased interference in other applications by bringing large amount of data without reusing it.

In some benchmarks, interference misses are over 50% of the total misses. Therefore, significant degradation in performance compared to tasks running by themselves is observed. Even in best cases, 10% of the misses is due to interference.

To eliminate the interference, partitioning the cache is proposed in this paper. Moreover, some tasks require less cache sizes. With configurable caches, disabling some partitions of caches is provided which also results in less power consumption. To partition caches, the cache is divided into a set of rectangles in which, columns represent ways and rows represent sets. Partitioning is accomplished through a static off-line approach where for each task there is a partition with the goal of optimizing the hit rate for each task, i.e. maximizing the cache utilization.

There are three conditions to solve this optimization problem. The first one specifies that partitions are non-overlapping. The second one ensures that the sum of all the cache partitions does not exceed the total cache size. The last condition is that the partition is implementable by the underlying configurable cache. This condition has an exponential complexity. For small systems it is feasible to solve it but for more tasks the complexity rises quickly. Therefore, a heuristic approach is proposed in this work. The heuristic set the smallest partition to all the tasks and adding them to an active list. Then the size and associativity are increased until the utility meets the desire value. At this point the task is removed from the active list.

The relaxed partitioning uses the fact that in many systems only a subset of tasks must meet the real-time deadlines and some tasks can occur offline. Therefore, a subset of non-critical tasks are treated as a single task and mapped to a single cache partition.

Results of different configurations suggest that larger cache size allows partitions to be closer to the point of marginal gains from the increased size.

2.4.4.2 A real-time capable first-level cache for multi-cores

A well-known problem in multi-core systems is simultaneous access to the same memory. In Non-Unified Memory Access (NUMA) architecture, the global shared memory is distributed among the cores. Cache Coherent NUMA (cc-NUMA) needs the shared data to be valid. Several methods such as bus-snooping and directory- based protocols assure the validity of shared data. However, they incur additional memory overhead. Moreover, for both shared and non-shared data, coherence transactions are regularly conducted, creating a high overhead. In case of WCET, possibility of invalidating a cache line by other cores, prohibits a tight estimation.

The idea of On-Demand Coherent Cache (ODC^2) [65] is to eliminate the interference between cores caused by coherence messages. The shared data is hold as long as needed and the possibly modified shared data is re-loaded at the time it is used. In order to enable this, two properties should be defined:

- Accesses to shared data is enclosed by critical regions and locks. Thus, a particular portion of shared data can be accessed by only a single core at any time.
- Cache lines cannot contain both private and shared data in parallel.

Coherent accesses are in case of shared data, otherwise no coherence is implemented. This is performed with two different cache modes: normal mode and shared mode. In normal mode, no accesses to the shared data is allowed. Hence, cache controller acts with no coherence functionality.

In the shared mode, additional information inside a cache line are used. If there is a cache miss in the shared mode, loaded cache line is marked with a shared bit. This indicates the possibility of shared data on this line. After deactivation of the shared mode, other memory operations are stalled and the restore procedure is performed. In a write-back strategy, all cache lines with the shared bit marked are flushed back to the main memory. Then all shared cache lines are invalidated. The activation and deactivation of the shared mode is triggered with load and store to specific addresses.

The shared data accesses need to be protected by critical regions. Since entering and leaving the share mode are directly related to the beginning and end of the critical region they can be integrated with the lock and unlock functions of the system. The real-time capability of this mechanism can be explained with the fact that ODC^2 prevents any external modification of the cached data.

Therefore, no simultaneous access from multiple cores to the same share data emerges. In the event of a read burst from the main memory due to a cache miss a write-back from the processing core is needed. Using ODC^2 this is eliminated. Moreover, there is no intervention forced by other cores to write back the cache line modified by a core.

The performance of ODC^2 can be discussed for the two modes of access. Inside the critical region all accesses are treated as shares accesses (even the private accesses). When entering the critical region, there is no shared data in the cache. Therefore, all the first accesses are misses. When leaving the critical region, the private data is invalidated wrongly. Therefore, it needs to be reloaded outside of the critical region (if re-used). The high amount of memory access caused by invalidation may have a negative effect on the performance. Outside of the critical region, only private data can be accessed. Therefore, there is no need for cache coherence mechanisms.

The ODC^2 method is evaluated on SoCLib, a simulation platform for on chip multi-core systems compared to MESI method. The number of bus accesses in ODC^2 is raised compared to MESI. This is due to invalidating the data which leads to cache misses. In general, higher number of shared memory accesses in ODC^2 is not counterbalance by coherent transactions of MESI. But it is slightly higher and is acceptable for the benefit of predictable timing behavior.

We introduced the dedicated caches concept in this section. In the next section, we investigate several designs using dedicated caches for improving the WCET of the cache.

2.5 Splitting the memory accesses to dedicated caches for improving the timing-predictability

Lundqvist and Stenstrom [53] propose distinguishing memory accesses of predictable and unpredictable data structures. This way, an unpredictable memory access can be tagged as non- cacheable. Therefore, interference with the predictable data cache is avoided. Assuming a memory hierarchy with a single level, an unpredictable memory access incurs a miss penalty equal to accessing the memory. If the unpredictable memory access is cached, the WCET includes two cache misses, one for the cache miss and one to replace the missing block. Therefore, this method improves the WCET by a factor of two. Therefore, different data structures fall into one of the two categories of predictable or unpredictable accesses. This generally depends on the WCET estimation method. For example, arrays which are accessed by non-regular strides and are not input

data dependent, are in theory predictable. However, some estimation methods may consider it as unpredictable due to lack of complex analysis of data dependency. Results of this work show that in five out of seven benchmarks, all the data structures are predictable which leads to a better WCET. For the other two benchmarks, most of the accesses are predictable.

In the following we summarize some methods for treating different data references in the data caches differently. These methods are generally trying to improve the average case performance. However, they can shed a light on how splitting the data caches can improve the performance of the systems and give us insights to use the idea of split data caches for improving the WCET as well.

Schoeberl et al. [77, 86] propose splitting the data cache for different types of data and perform analysis individually for each area. By splitting the data cache, access to unknown address of heap memory does not affect the static analysis of other data.

M Huang et al. [37] argue that smaller caches consume less energy and propose to break the L1 cache into smaller structures to reduce the power consumption. The split the data cache to a Specialized Stack Cache (SSC) and a Pseudo Set-Associative Cache (PSAC). The SSC resides in parallel to the data cache.

The compiler usually inserts substantial amounts of spill codes during register allocation. Cooper and Harvey [14] use a small random access compiler- controlled memory to hold these spilled values. Using this method, most of the memory traffic due to compiler-inserted instructions is eliminated.

Gonzalez et al. [31] propose a dual data cache, including a spatial cache and a temporal cache. A locality prediction table, including the information related to the load/store instructions, predicts the type of locality for each memory access. The main advantage of this method is its ability to decide on temporal or spatial locality of vectors as well.

Milutinovice et. al [55] propose data tagging to separate data with temporal locality from data with spatial locality. This technique eliminates fetching the entire block into the temporal cache. Therefore, a smaller prefetch buffer for spatial data caching.

Calder et al. [12] propose a compiler-based data placement method called Cache-Conscious Data Placement (CCDP) to reduce the frequency of data cache misses. They profile the programs and use the profiling information for heuristic data placement to decrease the inter-object conflict on the cache. They show for and 8K data cache, the CCDP results in 30% reduction in miss rate.

Tyson et al. [91] propose a dynamic approach for deciding which data items should be cached based on the address of the load instructions generating the request for caching. They assign several bits to every load instruction and use a table similar to a branch prediction table to decide whether the load instruction causes misses and use this information to mark the instruction as non-cacheable.

Johnson et al. [40] suggest bypassing the cache for infrequently accesses data when it results in replacement of more frequently accessed data. The divide the program to a number of blocks with uniform access frequency. Moreover, they use a table (the memory access table or MAT) to keep track of the access frequency of each block. When a miss happens in the cache, the block with less access frequency (according to the table) is replaced in the cache. They show that using this method some benchmarks can gain up to 12% speedup.

The LSMCache proposed in [74] uses three cache modules that each exploits a different type of locality. One for spatial locality, one for temporal locality and the third one exploits both spatial and temporal localities as a standard cache does. Sgnchez et al. use a static locality analysis proposed by [101] to determine where the fetched data should be placed. The result of the analysis is used to put a hint in each load instruction. When the same data resides in several caches, all the containing caches are updated. Results of this work show less miss rate (about 0.4) for caches with smaller sizes.

Rivers et al. [69] propose dynamically redirecting the accesses to non-temporal data blocks to a buffer separate from the main cache, in numeric-intensive applications. The architecture of non-temporal buffer consists of a detection unit and a data storing unit. When the blocks in the main cache are evicted, if they have not been referenced during their life on the main cache, they are marked as non-temporal. When these blocks are accessed again they are allocated on the buffer for non-temporal data. The results of this work show performance improvements over the conventional direct mapped caches.

Prvulovic et al. [62] split the cache based on the spatial and non-spatial locality. For detecting the spatial accesses, they assign four bits to the spatial subcache. They assign an access bit to each of the words in a block. When the word is accessed, the bit is marked. When less than two of the bits are set on a block eviction, the block is considered as non-spatial. When two or more of the sub-blocks of the same block are present in the non-spatial subcache, the block is marked as spatial and cached in the spatial subcache. Results of this work show significant improvements compared to a conventional cache.

Utilizing reuse information for efficient management of caches is presented in [70]. Authors evaluate two different methods based on 1) the effective address of the data being used [69] and 2) the program counter of the memory instruc-

tions [91]. The method using the program counter, marks data as cacheable and non-cacheable based on their access patters. The scheme based on the effective address partitions the accessed to frequently and infrequently accessed. Authors use a variation of the MIPS instruction set architecture (ISA) for their experiments. This work shows that the method based on the effective address performs better than the method that uses the program counter.

All of these works show promising results for split data caches. In spite of these results, novel architectures only exploit unified data caches. Using a single cache for different data types may results in better performance for the average-case. For real-time systems, though, accurately predictable data accesses are of more importance.

2.6 Scratchpad memories for improving the WCET and time-predictability

Scratchpad memories have been proposed as an alternative to caches in embedded systems [21]. The scratchpad memory is a special SRAM (static random-access memory) near the processor. Processor can access the scratchpad memory with very fast access cycles (one or two processor cycles). The access method to the scratchpad memory is address mapping, i.e. using explicit load/store to appropriate addresses. Thus, reducing energy consumption and access latency independent of the proceeding memory access patterns, are advantages of this type of memory. The challenge of using the data scratchpads is determining the type of the data to put into them. Firstly the small size restricts the data that can be put to the scratchpad. Moreover, the explicit accesses by load/store instructions needs the content of the scratchpad to be determined by the developer or the compiler. Thus, the form of the data supported by the scratchpad is restricted. However, frequently accessed data such as stack data and local variables can benefit from the fast access of the scratchpad [20].

Whitham and Audsley [98] propose scratchpad memory management unit (SMMU) as an alternative solution to pointer invalidation and pointer aliasing. The SMMU makes memory access operations time- predictable and does not need pointer analysis in the whole program. The SMMU solves the problem of aliasing and invalidation by separating logical and physical addresses. The SMMU allows objects to be resident in unchanged logical addresses even when they are moved to new physical addresses.

Lee et al. [48] propose to divert all the references to the stack data to a stack value file (SVF) instead of L1 data cache. Each memory location maps to a

register in the SVF based on the lowest address bits. When the stack pointer value changes, data from the first level cache moves to the SVF. For a stack machine, such as the Java virtual machine, the two top elements of the stack cache can be implemented as dedicated registers, which can be directly accessed for the ALU operations [81].

Bai et al. propose the limitless stack data management that can run any application on the limited local memory in [8]. They resolve the problem of pointer referencing to functions evicted from the local memory. In this work, a pointer is set to a global address rather than a local address. A function s2p converts the address by finding which function the pointer belongs to. Then the s2p function computes the offset of the pointer variable from the start of the frame in the local memory. Finally, the s2p returns the global address of the pointer by subtracting the offset from the global start address of frame of function. This technique can run any application on the limited local memory with the least amount of stack space.

Circular Stack Management (CSM) [43] is a software technique to keep the active stack data on a Scratchpad Memory (SPM). This method does not depend on profiling information. Moreover, CSM does not need to know the SPM size until run-time. This work introduces software SPM Manager (SPMM). The SPMM uses a function table with function addresses and their stack frame sizes. SPMM calls a specific function to check if there is enough space in the SPM. SPMM evicts a function from the limited local memory and move it to the global memory when there is not enough space in the SPM. After the function returns, SPMM checks whether return is to a valid stack frame.

Lu et al. optimize the CSM approach in [52]. The optimizations use 1) increasing the granularity of the stack data management. Increased granularity causes less latency of accessing the main memory, 2) eliminating calling the special functions that move data when there is space for the stack frame of the callee function and 3) performance of minimal work during the stack management. The authors define a weighted call graph of function calls. They propose a heuristic method and ILP to solve the problem of finding the optimal solution of cutting sets in the graph. Each set is corresponding to a pair of functions managing the transfer with the main memory. The goal of the methods is to find the minimum cost of transfer with the main memory.

In [60] authors propose to change the mapping address of SPM dynamically according to the stack pointer. They propose management policy for two cases. 1) Accessing the stack region over SPM region. 2) Accessing the stack below the SPM region. In both cases a fault handler copies the data between the SPM and DRAM.

Park et al. [59] use the memory management unit (MMU) for a dynamic address mapping of SPM. Their method requires no architectural modification or compiler assistance and generates permission faults when access to the stack region is outside the SPM. Dominguez et al. [23] introduce a method to allocate stack data of recursive functions to scratchpad memory. Profiling information helps to place the most commonly occurring stack depths in the SPM.

Wehmeyer et al. [96] explore the effect of using scratchpad memory on the WCET. They use *encc* compiler to count the number of accesses to variables and assign the memory objects to the scratchpad memory based on their access frequency. They formulate this as an ILP to select the elements with the highest benefit in terms of energy consumption. Authors show 43% and 58% improvement in the average case and worst case execution times respectively.

Ditzel et al. [22] suggest using the top element of the stack in high speed registers. They determine if access to the data should reside in the stack registers based on its address. Two pointers define the range of the stack accesses.

In [42] authors argue that static cache miss analysis techniques can be fragile and are dependent on memory layout changes. They suggest a data allocation scheme tries to minimize the inter-data cache pollution misses. The authors use both the SPM and data cache synergically for different data categories. For allocating the data to the SPM authors suggest a heuristic based on the number of intrinsic or interference miss counts. Results of this work show significant reduction in both the intrinsic and inter-task (interference) cache misses.

2.7 Hardware support for context switching

We now review some existing work using hardware support to optimize context switching. [90] and [56] introduce hardware support to optimize the context switching in real-time systems, but at the register file level. For instance, [90] uses dedicated hardware for scheduling threads in an SMT-based processor. The hardware scheduler is also able to save/restore the registers of a thread to a special on-chip memory, the *Thread Control Block* (TCB). The TCB requires two separate ports, in order to eliminate any interference from parallel accesses to the TCB from the running program and the hardware scheduler. Our work is orthogonal, as we optimize context switching at the cache-level. The use of virtual stack caches furthermore opens new opportunities such as context saving to off-chip memory, which, according to our experiments, appears to be feasible without additional hardware costs. Others, such as [88], optimize the average cost of context switching, but due to lacking predictability these methods are

unsuited for real-time systems.

Treating data from the program's stack differently than the non-stack data was already proposed in [58], but for reducing dynamic energy. They introduce an implicit and an explicit stack data cache. The implicit stack data cache limits the stack data to reside in specific locations (ways) of the regular data cache. In the case of the explicit stack data cache a separate data cache is reserved for stack data. The use of standard caches makes this approach amenable to standard CRPD analysis techniques. However, the worst-case behavior for such a design was, so far, not evaluated.

2.8 Summary

In this chapter we have presented basic concepts related to time-predictable architectures. We have provided an overview of state of art time-predictable architectures and their specifications. We have described the Patmos processor which is part of the T-CREST architecture, as our target for time-predictable stack caching implementation. Thus, we have presented the main architectural and compiler features that make Patmos a time-predictable embedded processor.

We looked into data caches as important part of embedded systems, the different replacement policies used in caches and how they affect the WCET of caches and hence the system using them.

We continued the discussion by explaining different methods of improving the WCET in caches and how splitting different memory accesses to dedicated caches can improve the performance of the data caches. Moreover, we briefly discussed the WCET of data caches in systems with shared caches and how the interference between caches can be a bottleneck for a precise WECT estimation.

We reviewed the state-of-the-art work on using the scratchpad memories as fast replacements for data caches and different methods of allocating the data on scratchpad memories to improve the WECT.

Finally, we looked into some examples of hardware support for context switching as we provide a hardware design for reducing the context switching in this thesis as well.

Chapter 3

Time-predictable Stack Cache

In this chapter we present a time-predictable stack cache with implementation details. A time-predictable stack cache in real-time systems, avoiding interference with the heap data accesses, can improve the time-predictability of the data cache. Explicit instructions implemented in hardware manage the stack cache. These instructions guarantee the loads and stores instructions accessing the stack cache are always hits and therefore improving the stack cache's time-predictability. Moreover, we describe integration of the stack cache with the Patmos processor and adaptation of the processor's pipeline stages to support the integration.

This chapter is structured as follows: in Section 3.1 we describe the idea of the time-predictable stack cache as a method to simplify the problem of timing analysis in data caches. In Section 3.2 we describe the detailed implementation of the stack cache in hardware and porting the implementation to the Patmos processor [87]. In Section 3.3 we discuss some of the aspects of the stack cache in more details. We talk about the WCET of the stack cache in 3.4. We finish the chapter by highlighting the main parts of the stack cache in the summary of Section 3.5.

This chapter is based on the following published article: *A Time-Predictable Stack Cache* [3].

3.1 Stack cache

The stack frame of a program's (sub-)routine typically contains information of the return addresses, saved register values, as well as function-local variables and data structures. This type of data is frequently accessed and thus benefits from caching. Moreover, we can statically determine the addresses of stack data when there is no dynamically sized allocation data on the stack [67]. Therefore, we propose a hardware managed stack cache, dedicated for the stack data to assists the time-predictability of the design. We call this cache, the *stack cache*.

The stack cache reduces the number of loads from the data cache and thus decreases the number of slow main memory accesses. Additionally, the stack cache eliminates the long latency stores to the write-through data cache.

Moreover, splitting data caches by their access pattern, eliminates the interference of the heap allocated with the stack allocated data. Thus, split data caches allow for analysis of the worst-case behavior of these data types accesses independently, resulting in more accurate WCET.

Furthermore, the local variables in a C function or Java methods are undefined outside the scope of the function or the method. Therefore, after a function return there is no need for writing back the stack data to the main memory and thus a cache dedicated to the stack data eliminates the need of consistency with the main memory.

3.1.1 Stack cache instructions

The stack cache is a special ring buffer manipulated with two pointers: *stack top* (`sc_top`) and *memory top* (`m_top`). These pointers are full length hardware registers specifically dedicated to the stack cache. The `sc_top` is the address of the top of the stack data and the `m_top` is the address of the top element in the main memory. The initial values of the `sc_top` and `m_top` are the same. Difference between `sc_top` and `m_top` is the occupied space in the stack cache. The stack cache is managed in 32-bit words and thus the pointers count in 32-bit words as well. For a stack cache of size of 2^n words, when addressing the stack cache, only the lower n bits are used. Therefore, when accessing the stack cache, we mask the `sc_top` to point to the appropriate address range. Assuming a stack cache of size `SC_SIZE`, the mask, `SC_MASK` equals to `SC_SIZE`-1.

Three dedicated stack cache instructions manipulate the stack cache. The Patmos ISA is extended with these instructions to support the stack cache. The

stack cache is managed within programs scopes. Since function calls allocate space on the logical stack of a program, we explain the stack cache instructions using function calls. Although, the same explanation applies to any other program scope that uses the stack instructions for allocating space on the stack cache. Upon function entry, the reserve (`sres`) instruction allocates a number (parameter of the `sres` instruction) of words on the stack. Before a function returns, the free instruction (`sfree`) frees the allocated stack space of the function. The ensure instruction (`sens`), ensures a valid stack cache frame after a function returns. Typed load and store instructions access the reserved space on the stack cache.

Assuming a stack cache growing downwards, and the argument of stack cache instruction to be `x`, and `sc` and `mem` representing the stack cache and the main memory, implementation of the stack cache instructions are as follows:

Reserve The `sres` instruction moves the `sc_top` downwards `x` words. In case there are not enough free words in the stack cache to reserve, *spill* operation transfers part of the stack cache data to the main memory to free the stack cache for the words required by the reserve instruction. The `m_top` is decremented according to the number of transferred words to the main memory to point to the top element of the stack cache in the main memory. Figure 5.2 shows the implementation of the `sres` instruction in pseudo code.

```
void reserve(int x) {
        int nspill, i;
        sc_top -= x;
        nspill = m_top - sc_top - SC_SIZE;
        for (i=0; i<nspill; ++i) {
                --m_top;
                mem[m_top] = sc[m_top & SC_MASK];
        }
}
```

Figure 3.1: The `sres` instruction provides `n` free words in the stack cache. It may spill data into main memory.

Free The `sfree` instruction moves the `sc_top` upwards `x` words. When the `sc_top` value is greater than the `m_top`, the data between the `m_top` and `sc_top` in the main memory is no longer valid. Therefore, the value of `m_top` is increased to the `sc_top` to point to an address with valid data. Figure 5.3 shows the implementation of the `sfree` instruction in pseudo code.

```
void free(int x) {
    sc_top += x;
    if (sc_top > m_top) {
        m_top = sc_top;
    }
}
```

Figure 3.2: The `sfree` instruction drops `n` elements from the stack cache. It may change the top memory pointer `m_top`.

Ensure The `sens` instruction checks the number of valid words in the stack cache. In case the number is less than the argument `x`, the *fill* operation transfers the required number of valid words from the main memory to the stack cache. These words are previously spilled to the main memory by a `sres` instruction. The `m_top` is decremented according to the number of transferred words to point to an address with valid data. Figure 5.4 shows the implementation of the `sfree` instruction in pseudo code.

```
void ensure(int x) {
        int nfill, i;
        nfill = x - (m_top - sc_top);
        for (i=0; i<nfill; ++i) {
                sc[m_top & SC_MASK] = mem[m_top];
                ++m_top;
        }
}
```

Figure 3.3: The `sens` instruction ensures that at least `n` elements are valid in the stack cache. It may need to fill data from main memory.

Load and Store In the Patmos processor, typed load and store instructions are used to access different types of memories, e.g. load and store instructions to access the main memory. To access the stack cache, typed load and store instructions for the stack cache are defined as well.

Due to execution of the `sens` instruction before entering a program scope, the data belonging to that scope is always present in the stack cache. Therefore, load and store instructions accessing the stack cache are always hits. The implementation of the load and store instructions is shown in Figure 5.5. Note that the address of a load or store instruction, is calculated with adding the current

sc_top value to the argument of the load or store instruction. As we mentioned earlier, the calculated address is a full length address. Thus, to access the stack cache, we mask the upper bits of the address with the SC_MASK to generate an address within the stack cache address range.

```
// load one word from the stack cache
// addr is a plain main memory address
int load(int addr) {return sc[(addr + ST) & SC_MASK];}
// store one word into the stack cache
// addr is a plain main memory address
void store(int addr, int val) {sc[(addr + ST) & SC_MASK] = val;}
```

Figure 3.4: Pseudo code for the load and store instructions.

To make the functionality of the stack cache clear, next, we explain the execution of the stack cache instructions. We identify the value of the sc_top and m_top and the content of the stack cache and main memory through an example.

3.1.2 Function call example

We illustrate the stack cache functionality with an example of three function calls: Function A calls function B, and function B calls function C. Figure 3.5 shows the mapping of the stack cache to the main memory. The blocks A, B, and C in the figure on the left box are the stack frames for functions A, B, and C respectively. The stack mapping shows the total stack usage without a stack cache. We assume the stack cache cannot hold the contents of all the three functions, i.e. the size of the stack cache is smaller than sum of the sizes of stack frames of these three functions. The middle box (S$) shows the content of the stack cache. The box on the right side shows the main memory content. The main memory and stack cache states are captured at a time when the program is in running function C.

The program in this example starts with an empty stack cache. After entering function A, its stack frame is allocated using a reserve instruction. This stack frame is represented as block A in Figure 3.5. During execution of function A, function B is called. Therefore, a reserve instruction allocates function B's stack frame. The reserve instruction checks whether the required space for function B's frame is available in the stack cache. Assuming that there is not enough free space in the stack cache, the reserve instruction spills part of the stack frame of function A to the main memory. Calling function C (Figure 3.5) spills the rest of the function A's stack frame and a part of function B's stack frame to the

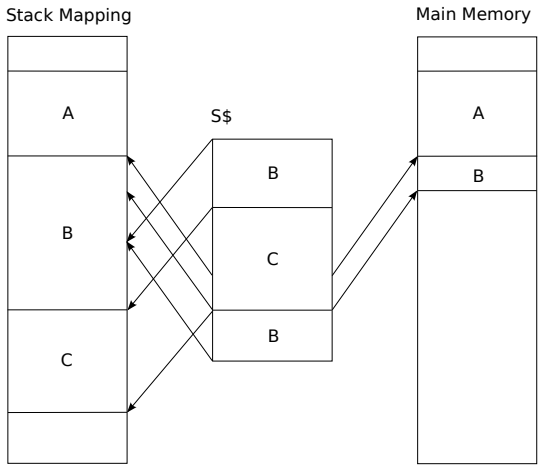


Figure 3.5: Stack mapping, the stack cache content, and the spilled stack area.

main memory. Before returning from function `C`, a free instruction deallocate function `C`'s stack frame. When returning from function `C` to function `B`, an ensure instruction reloads the spilled parts of function `B`'s stack frame back into the stack cache. Similarly, before returning from function `B`, a free instruction deallocate function `B`'s stack frame. The program returns to function `A`, and the whole stack frame of `A` is reloaded from the main memory. Before returning from `A`, all the allocated space on the stack cache is freed and the stack cache becomes empty again. Figure 3.6 shows the function calls and the stack cache instructions executed during the calls.

3.2 Hardware implementation

Hardware-description languages like Verilog and VHSIC Hardware Description Language (VHDL) do not provide modern software languages' powerful abstraction facilities. Therefore, hardware design with these Hardware Description Language (HDL)s often leads to low productivity and makes it difficult to reuse components [7]. For these reasons we chose to implement the stack cache in *Chisel* [7]. Chisel is an open-source hardware design language developed at UC Berkeley. Chisel supports advanced hardware design with highly parameterized generators and is embedded in the Scala programming language. Chisel can generate fast cycle-accurate C++ simulators, or Verilog codes suitable for FPGA emulation.

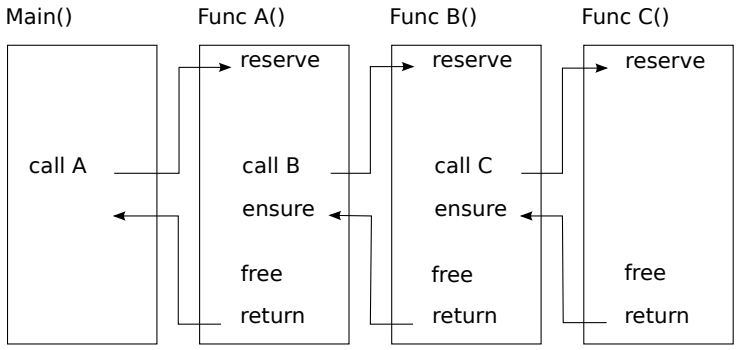


Figure 3.6: Three function calls and stack cache instruction executed on entering and exiting each.

Table 3.1: `OCPburst` signals and their functionality

Signal	Description
`MCmd`	Command from the master (read or write)
`MAddr`	Address from the master, lowest two bits always 0 (word address)
`MData`	Data from the master for write command, with 32 bits length
`MDataByteEn`	Byte enables from the master for sub-word writes, 4 bits
`MDataValid`	Signal that data is valid, 1 bit
`SResp`	Response from the slave
`SData`	Data from the slave in response to read command, with 32 bits length
`SCmdAccept`	Slaves signals that it accepted the command, 1 bit
`SDataAccept`	Slave signals that it accepted the data, 1 bit

As we mentioned in chapter 2 for the connection of the Patmos processor to the main memory, a subset of the OCP [5] standard for the hardware interface is used. The OCPcache [84] variant is generated for communication between the pipeline and the caches. Therefore, we use the OCPcache as the interface between the stack cache and Patmos. Moreover, caches in the Patmos access the main memory through bursts using the OCPburst [84]. Thus, for communication between the stack cache and the main memory we use OCPburst [84] as well. Table 3.1 explains the signals of the OCPburst. This table shows two sets of signals, *master* and *slave*. For transfers between the stack cache and the main memory, the former is the master and the latter is the slave. [84] depicts the details of communication between the master and slave.

3.2.1 Stack cache controller

The stack cache is a set of memory blocks, each with one read and one write port. However, these blocks form a unified memory block. Separation of the memory blocks enables us to access (i.e. read or write) them individually and facilitate our design for a byte accessible processor, e.g. Patmos. To manage the reading and writing of these blocks and the interactions with the processor's pipeline, we developed a stack cache controller. The stack cache controller performs the operations requested by the processor's pipeline and the spilling and filling operations whenever needed. In the following we explain the details of the design of the controller. As we explained in 3.1.1, there are special instructions to manage the stack cache. Moreover, there are special instructions that can set and get the values of the two stack cache pointers. The stack cache controller receives these instructions in the memory stage of the processor pipeline. These instructions and the controller's operation for each of them are as follows:

- sc_OP_NONE: No action is required, i.e. none of the stack cache instructions or special instructions to access the pointers are issues in the processor's pipeline.
- sc_OP_SET_ST: Assigns the operation's operand to the `sc_top` pointer (i.e. updates the `sc_top`).
- sc_OP_SET_MT: Assigns the operation's operand to the `m_top` pointer (i.e. updates the `m_top`).
- sc_OP_ENS: Computes the required `m_top` pointer, and checks if filling is needed.
- sc_OP_RES: Computes the new `sc_top` pointer and the required `m_top` pointer, checks if spilling is needed.
- sc_OP_FREE: Moves the `sc_top` pointer upwards.

The stack cache controller is a state machine with six states. The state machine manages the spill/fill operations. Moreover, it keeps track of the number of spilled/filled bytes and adjusts the two pointers to the top of the memory and stack top when necessary. In the following we explain each state's function in more details. Figure 3.7 shows the stack cache's state machine.

idleState This state is where the controller starts its operation or when no operation is needed. It computes new values of the stack cache pointers depending on the instruction type that pipeline passed to the controller. Moreover, it

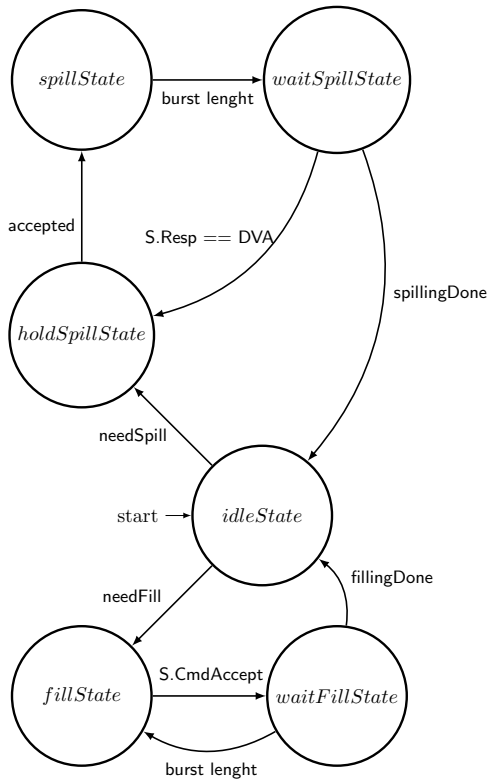


Figure 3.7: Stack cache controller's state machine.

checks whether spilling or filling operations should happen. The state machine goes to either of the *fillState* or *holdSpillState* states for a fill or spill operation respectively.

holdSpillState This state generates an OCP write request and waits for the slave (i.e. the main memory) to respond to the write request. Moreover, the transfer address is set. Then, in the next possible cycle the right data is read from the stack cache and transferred to the main memory. When the slave accepts the write request, address is incremented (for reading the next data) and state is changed to the *spillState*.

spillState This states checks whether a write burst transfer to the main memory is possible. Firstly it generates an OCP write request and checks if a write command has been accepted. If the command is accepted by the slave (i.e. the main memory), it reads the next data element from the next address and changes the state to *spillState*, otherwise it holds the previous address and stays in this state.

waitSpillState This state transfers data to the OCP bus (to be transferred to the main memory). It reads the next data element from the stack cache's memory and increments transfer address. On each write (read) command the maximum number of the transferred words is defined by the burst length. Spilling continues until the number of transferred words reaches the burst length.

fillState This state generates an OCP read request command and waits until it is accepted. When the command is accepted it goes to the next state (i.e. *waitFillState*) for the actual filling operation.

waitFillState This state first checks whether any data is left to be written to the stack cache and depending on that, it writes the data to the stack cache memory. Finally, it checks whether the burst length is reached to start a new read from the main memory or it simply finishes the filling when all the required data has been read from the main memory (i.e. the read address is equal to `m_top`).

3.2.2 Integration with the Patmos processor

Figure 3.8 shows different pipeline stages and different components of the Patmos processor (including the stack cache in the memory stage). For integration of the stack cache to the pipeline we need to extend the processor to include the stack cache's instructions. Moreover, the stack cache controller needs to communicate with the pipeline. In the following, we explain the extensions to the pipeline for the integration.

3.2.2.1 Decode stage

According to the algorithms introduced in 3.1.1, three instructions manage the stack cache. We extended the decode stage to support these instructions.

3.2.2.2 Execution stage

The two stack pointers in the Patmos processor are special registers and two special instructions *mfs, mts* read and write these registers. In the execution stage of the processor, the *mts* instruction writes the value of its first operand to either of the `sc_top`or `m_top` (depending on the second operand of the instruction). Moreover, the *mts* instruction writes the value of the `sc_top` or `m_top` forwarded from the stack cache controller to the execution stagethe *mfs* instruction reads the value of the `sc_top` or `m_top` forwarded from the stack cache controller to the execution stage.

3.2.2.3 Memory stage

The stack cache design in the memory stage comprises the stack cache itself and the stack cache controller. Since the stack cache transfers data to/from the main memory, in the memory stage, we need to differentiate between these two:

Data Transfer with the Pipeline Patmos processor uses the OCPcache to communicate with the caches. Therefore, we extended the multiplexer in the memory stage to transfer the OCPcache commands between the stack cache and the pipeline whenever the typed load or store is from/to the stack cache.

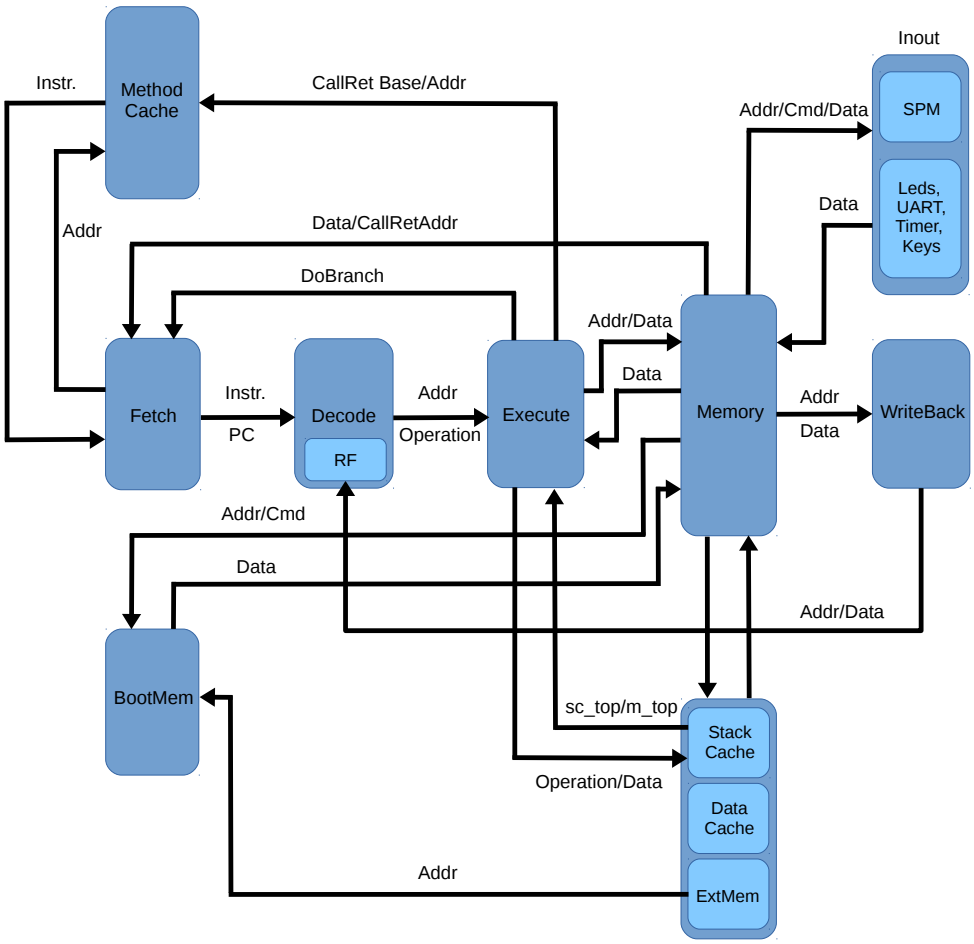


Figure 3.8: Block diagram of Patmos components.

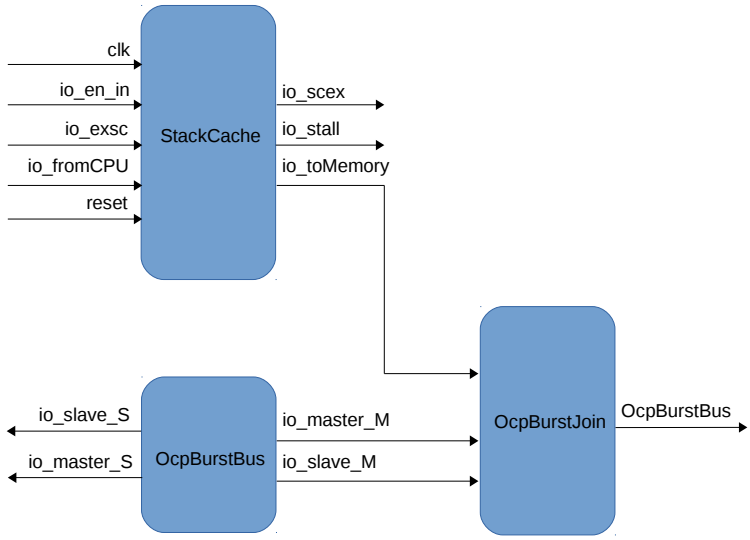


Figure 3.9: Join of the read request from the stack cache with read bus.

Data Transfer with the Main Memory Since the Patmos processor's data cache sends and receives data to/from the main memory as well, it is important to merge the signals transferred between either the stack cache or data cache and the main memory (i.e. which of these components should send/receive command or data to/from the main memory at any given moment). Therefore, we extend the merge component in the Patmos processor. Figure 3.9 shows the signals of the stack cache and the merge component. The merge component is a simple multiplexer, passing the signals of the input with a valid (i.e. non-idle) request (read or write). For a read from the main memory, when it sends the requested data to the pipeline, the data is passed to the corresponding caches. Patmos processor includes a write combine buffer as well. For spilling from the stack cache, we bypass the write combine buffer, since the stack cache's data should go to the main memory on spilling without any interference.

3.2.2.4 Write back stage

The stack cache does not modify the write back stage. The only effect of the stack cache on the write back stage is that the stall signal disables writing to the register file during the stall cycles.

3.3 Discussion

The organization of the stack cache implies some limitations:

- The maximum size of stack data accessible at any moment is limited to the size of the cache. The stack frame can be split, such that at any moment only a subset of the entire stack frame has to be resident in the stack cache, or a shadow stack frame in global memory can be allocated.
- When passing pointers to data on the stack cache to other functions it has to be ensured that: (1) the data will be available in the cache, (2) the pointer is only used with load and store operations of the stack cache, and (3) the relative displacement due to reserve and free operations on the stack is known. Alternatively, aliased stack data can be kept on a shadow stack in the global memory without restrictions.
- The stack control operations only allow allocating constant-sized junks. Computed array sizes (C 90) and alloca with a computed allocation size have to be realized using a shadow stack in global memory.

3.3.1 Alignment of burst transfers

The stack cache operates on words and thus needs to align transfers according to the main memory's requirements. Therefore, we need to perform the alignment for transferring data to the main memory. Since there is no way to know whether an address is aligned before accessing it, we need to assume all the addresses are unaligned. Transferring invalid data to align all the accesses leads to less optimal utilization of the main memory's bandwidth. In the chapter 4 we propose a method to fix this issue.

3.4 WCET analysis

As we mentioned in the background chapter, in a standard data cache each load or store access to the data cache can result in a cache miss and the program analysis should take it into account. The compiler generated `sres` and `sens` instructions in the stack cache, mark program points in which an interaction with the main memory (spill/fill operations) *might* occur. Therefore, any other access to the stack data is a guaranteed hit. A simple, but very conservative, WCET estimation for the stack cache can assume spill and fill on every `sres`

and `sens` instruction respectively. Although, as we mentioned, any other access to the stack data is a hit. A simple improvement is to statically analyze the call tree and cut out all functions that are within a stack cache size from the call tree leaves. Such a part of the call tree needs to spill only one stack cache size, the rest of the stack `sres` and `sens` instructions need no spill or fill operations. A more advanced analysis may perform data flow analysis of the two pointers `m_top` and `sc_top` to find the exact spill and fill points and sizes.

3.5 Summary

Although caches can improve the average performance of systems, they are bottleneck to WCET analysis of real-time systems. A time-predictable stack cache in real-time systems can improve the predictability of the data cache, avoiding interference with the heap data accesses. In this chapter we proposed time predictable stack cache to split the stack data accesses to simplify the analysis of data caches. Stack data addresses are statically predictable and therefore splitting them simplifies the data cache analysis. We designed a cache solely serving the stack allocated data. Moreover, we integrated the stack cache to the time-predictable Patmos processor. The benefits of a cache dedicated to stack data are: Firstly, we can optimize the cache design for stack data in which transfer to/from the main memory happens on certain program points. Secondly, all the accesses to the stack cache besides these certain points are guaranteed hits, improving the WCET. It is important to note that a cache miss in the stack cache can only happen on a function return. A miss may happen because some blocks of the caller of this function are not existing in the stack cache, since the function that returned spilled them. However, the blocks of this function are not useful anymore, and can be invalidated. Then, the missing block in the stack cache can be over-written (filling) on the invalidated blocks. Since the program is already out of the scope of the returned function, there is no need to write these invalidated block to the main memory. Therefore, we can analyze the call tree of a program to know where these spills and fills happen and precisely estimate the WCET of the stack cache. We described the design and implementation of the time predictable stack cache in hardware. Moreover, we have integrated it with the time-predictable processor Patmos. In chapter 6 we present the evaluation of our design.

Chapter 4

Practical Improvements to the Stack Cache

In this chapter we present two different methods for optimization of the stack cache with implementation details. The first method, *lazy spilling* is based on the idea of avoiding spilling unnecessary blocks to the main memory by introducing a new pointer. This method thus improves the performance with very small hardware overhead. The second method, *block alignment* is based on the elimination of transfer of redundant data required for aligning the transferred data between the stack cache and the main memory. Moreover, we introduce a hardware extension to *virtualize* several stack caches in a shared memory, which allows us to quickly switch between these virtual caches. The preemption overhead can partially be hidden through this extension, which is profitable for the Worst-Case Response Time (WCRT) of the preempting task. This furthermore opens promising opportunities to save/restore virtual caches of preempted tasks during the execution of other tasks.

This chapter is structured as follows: in Section 4.1 we describe the idea of the lazy spilling stack cache for improving the performance of the time-predictable stack cache. We describe the implementation of the lazy spilling stack cache in hardware and porting the implementation to the Patmos processor. In Section 4.2 we explain the second optimization method for the stack cache. We start by motivating the idea and describe the implementation of the block-aligned stack cache in hardware. In section 4.3 we introduce the problem of

schedulability analysis and the stack cache's effect on this problem. Moreover, we describe the idea of virtual stack caching and explain the hardware implementation. Finally, we discuss the scheduling opportunities based on our proposed virtual stack caching method. We finish the chapter by highlighting the main parts of the these three methods in the summary Section 4.4. This chapter is based on the following published articles:

- *Lazy Spilling for a Time-Predictable Stack Cache: Implementation and Analysis* [73].
- *Alignment of Memory Transfers of a Time-Predictable Stack Cache* [72].
- *Efficient Context Switching for the Stack Cache: Implementation and Analysis* [4].

4.1 Lazy spilling stack cache

We introduced the stack cache in the previous chapter. In the next section we introduce an optimization for improving the performance of the stack cache's reserve instruction and hence improving the overall performance of the stack cache. Our optimization is due to the fact that in many cases, the actual data that a reserve instruction spills is exactly the same as the data already stored in the main memory. This is due to some situations when the stack cache repeatedly spills the data to the main memory, e.g., function calls in a loop. When a `sres` instruction causes spilling data from the stack cache to the main memory, one of these two cases happen: 1) subsequent store to the stack cache instructions write to the same addresses as the spilled words of the stack cache or, 2) subsequent store to the stack cache instructions do not write to the same addresses as the spilled words. Therefore, in case of the latter, the spilled data might not be modified between these accesses, i.e. it is not overwritten by other possibly spilled data. Thus, main memory and the stack cache contain the same data. Therefore, if another reserve instruction causes the spill of the same words to the main memory, the content of the main memory and the stack cache are already consistent and there is no need to transfer these words to the main memory. Another case where unnecessary spilling happens is when after a reserve instruction, reserve instruction spills the uninitialized cache data, i.e. the reserved space in the stack cache that is not used by a subsequent store instruction (no data is written to the reserved addresses). Therefore, we propose to keep track of the data that is coherent between the main memory and the stack cache. Thus, the reserve instruction can avoid the unnecessary spills of the coherent data to the main memory. Therefore, resulting in elimination of

extra cycles of transfer to the main memory and improving the performance. We show that this tracking can be realized efficiently using a single pointer, the so-called *lazy-pointer*. This pointer marks the beginning of the coherent data and thus needs to be updated by the stack control instructions as well as store instructions modifying the stack cache's content. In the following we motivate our idea with a detailed example.

4.1.1 Motivation

Figure 4.1 shows a function `bar`, that uses the stack cache to allocate a stack frame of two words (l. 2) on a stack cache with a size of 8 words, i.e., $|SC| = 8$. The allocated stack frame is freed before returning from the function (l. 16). The function executes a loop (l. 5–14), which repeatedly calls another function `foo`. We assume that function `foo` reserves 8 words and thus evicts the entire stack cache content each time it is called, i.e., its displacement is 8 words. Furthermore, we assume that on the start of the program no stack data has been allocated so far, i.e., the stack cache is entirely empty. The `m_top` and `sc_top` pointers are thus equal, pointing to the top of the empty stack at address 256. Clearly, `bar`'s `sres` instruction (l. 2) performs no spilling and simply decrements the `sc_top` by two. At this point, `m_top` points to address 256 and `sc_top` to 248 resulting in an occupancy of 2 words or 8 bytes ($256 - 248 = 8$). The `m_top` and `sc_top` pointers are not modified by the store and load instructions (l. 4, 7) and change after the function call to `foo`. At this point, a `sres` execution reserves 8 words. Simply decrementing `sc_top` by 32 is not enough in this case as the new value of `sc_top` ($248 - 32$) with an unmodified `m_top` would result in an

```
1  function bar()
2  sres 2
3  // store loop-invariant stack data
4  sws [1] = ...
5  loop:
6  // load loop-invariant stack data
7  lws ... = [1]
8  // displaces entire stack cache
9  call foo
10 // reload local stack frame
11 sens 2
12 cmp ...
13 // jump to beginning of loop
14 bt loop
15 // exit function
16 sfree 2
17 ret
```

Figure 4.1: `foo` evicts the entire stack cache, `bar`'s stack data is thus spilled on each iteration.

occupancy of 10 words (256 − 216) that exceeds the stack cache size. The two words allocated by `bar` are thus spilled, decrement `m_top` to point to address 248. Before returning from `foo`, its stack frame (8 words) is freed, simply by increment `sc_top` by 32 (8 words), which then points to the same address as `m_top` (248). This means that the stack cache is now empty. However, 2 words of `bar's` stack frame are still on the logical stack. More precisely, they are stored in main memory in the address range 248 to 255. The `sens` instruction (l. 11) executed next, is required to ensure that these two words are present in the cache for the load of the next loop iteration. The operation thus performs a cache fill and increments the `m_top` pointer to 256. The current cache state is thus identical to the state before the function call. For subsequent loop iterations, the stack cache state changes in the same exact pattern: 8 words are reserved by `foo`, causing the 2 words of `bar` to be spilled to the main memory, which are then reloaded to the stack cache. At this point the values of the respective stack slots are loop-invariant and do not change. Consequently, on every iteration the exact same values from the 2 words of the function `bar` are written (spilled) to the main memory while the respective memory cells already hold these values for all the loop iterations except for the first. The standard stack cache is obviously lacking means to avoid this redundant spilling of the stack data that is clearly *coherent* between the stack cache and the main memory. In the following we explain a standard method to avoid the problem presented.

4.1.2 Dirty bits

Standard cache designs often use a *dirty* bit to track cache entries with modifications. In a write-back cache, each cache line of the cache can contain an extra bit: the dirty bit. Each time the processor writes to the cache, the dirty bit of that line is set. When the cache controller needs to replace the cache content, it checks the dirty bits and updates the words with their dirty bit set in the main memory. Non-modified dirty bits indicate unchanged data and therefore, there is no need to write the corresponding data back to the main memory. This technique is, of course, applicable to the stack cache. A dedicated memory containing one bit per each word of the stack cache can keep track of the consistent data between the stack cache and the main memory to avoid spilling the unchanged words. Whenever the stack cache is full, the stack cache controller spills some words from the stack cache to the main memory. The processor executing next instructions, might write to the same address as the spilled words. Therefore, these words are not consistent with the main memory anymore and the dirty bit relevant to each of the changed words is updated. On the next spill of the same words to the main memory, the dirty bit of each word is checked. A dirty bit indicating the locally updated word, causes the stack cache controller

	Stack Cache Size (KB)	
Line Size (B)	1	2
4	8	256
8	128	64

Table 4.1: Total size of dirty bits memory in bytes for different line sizes.

	Stack Cache Size (KB)	
Line Size (B)	1	2
4	64	32
8	32	16

Table 4.2: Total size of dirty bits memory with one bit per burst transfer in bytes for different line sizes.

to spill the words to the main memory again. Otherwise, i.e. when the dirty bit shows no local changes to the words there is no need for spilling. Table 4.1 shows the size of dirty bits memory for different configurations of the stack cache and different line sizes.

Since the stack cache spill/fill operations transfer the data with the main memory with bursts of different lengths, each of the burst transfers needs a dirty bit corresponding to it. Since the burst transfers are unaligned, each time there is a data transfer with the main memory, all the words within one burst are transferred even if their dirty bit is not set. Therefore, for any number of words within a burst, one dirty bit suffices. Using one dirty bit per each burst reduces the number of bits by factor of burst size. Table 4.2 shows the number of bits for different cache configurations considering one dirty bit per each burst transfer for different line sizes. From the increased number of bits we can see that this method increases the hardware size of the stack cache considerably. More importantly, the static analysis of these various dirty bits complicates the computation of WCET estimates. An alternative solution is to introduce another pointer, which we call *lazy pointer* (`LP`). The `LP` keeps track of the address of the first coherent element on the stack. Keeping track of the coherent blocks using this pointer only requires an additional register and minor modifications to the stack store and control instructions. None of these changes introduces a substantial amount of hardware.

In the following we explain the situations where the `LP` value changes in the lazy spilling stack cache.

4.1.3 Lazy spilling

We propose to extend the original stack cache design with a new pointer, which we call *lazy pointer* (LP). The LP that keeps track of the stack elements in the cache that are known to be coherent with the main memory. In the original stack cache design the difference m_top − sc_top represents the amount of occupied space in the stack cache. Clearly, this value cannot exceed the total size of the stack cache's memory $|SC|$, as expressed by the following invariant:

$$0 \leq \texttt{m\_top} - \texttt{sc\_top} \leq |SC|. \tag{4.1}$$

In a stack cache with the lazy pointer, we introduce a new invariant. This invariant is based on the fact that the LP only refers to the data in the stack cache, thus the following invariant should always hold:

$$\texttt{sc\_top} \leq \texttt{LP} \leq \texttt{m\_top}. \tag{4.2}$$

As we mentioned, the LP points to the address of the first element on the stack cache that is coherent with the main memory. Thus, the LP divides the reserved space in the stack cache into two regions: (a) a region between the sc_top and LP and (b) a region between the LP and m_top. The former region defines the *effective occupancy* of the stack cache, i.e., it contains potentially modified data, not coherent with the main memory data. The data in the second region is coherent between the main memory and the stack cache. There are two potential cases when the LP should change:

- Stack pointer moving up: whenever the sc_top moves up past the LP or the m_top moves down below the LP, the LP needs to be adjusted accordingly to separate the two regions correctly.
- Store to the stack cache instruction: when a stack store instruction writes to an address above the LP, some data potentially becomes incoherent. Hence, the LP should move up along with the effective address of the store.

Next, we explain the details of the implementation of the stack cache's instructions supporting the lazy spilling.

4.1.4 Implementation

Following the above observations, we need to adapt the stack control instructions and the stack store instruction to support the lazy spilling. Note, however, that lazy spilling does not impact the `sc_top` and the `m_top` pointers, i.e., their respective values are kept identical to the original values during the execution of the stack cache instructions. We use the following constants from previous chapter 3.1.1: The stack cache size is assumed to be `SC_SIZE` and the mask, `SC_MASK`, equals to `SC_SIZE`-1.

`sres` n**:**
The stack cache's `sres` instruction decrements the `sc_top`. Moreover, the reserve instruction potentially spills some blocks of the stack cache to the main memory and thus might decrement the `m_top` value. To respect the invariant 4.2, we might require an adjustment of the `LP` value so that it stay below the `m_top`. Moreover, upon an initial reserve, the cache content is uninitialized. We can exploit this fact and treat this content as coherent with the main memory content. For instance, before the reserve instruction, when `sc_top` = `LP` , all the stack cache content is known to be coherent, which allows us to retain the equality `sc_top` = `LP` even after the `sres` instruction. Moreover, when the space allocated by the current `sres` covers the entire stack cache, all the data in the stack cache is uninitialized. In this case it is again safe to assume `sc_top` = `LP` after the reserve. Figure 4.2 shows the pseudo code of the `sres` instruction. The `lp_pulldown` variable is defined for checking the uninitialized stack data condition. As we see, a true value for the `lp_pulldown` indicates that the stack data is uninitialized and hence there is no need to spill it to the main memory. When the `LP` value gets more than the `m_top`, we adjust it to get the `m_top` value, since there is no need to spill stack data that is already spilled.

Apart from updating the `LP`, the spilling mechanism itself requires modifications. Originally, the `m_top` pointer is used to compute the amount of data to spill. However, when lazy spilling is used, the amount of data to spill does not depend on the `m_top` anymore. Instead, the `LP` has to be used to account for the coherent data present in the stack cache. With respect to spilling, the `LP` effectively replaces the `m_top` – hence the term effective occupancy for the region between the `sc_top` and the `LP`. As we can see in the loop in Figure 4.2, the `LP` value is used to compute the number of iterations (spills to the main memory).

`sfree` n**:**
The stack cache's `sfree` instruction increments the `sc_top`. The `m_top` may potentially increase as well. To satisfy the invariant from above, the `LP` is incre-

```
void reserve(int n) {
    int nspill, i;
    bool lp_pulldown = (sc_top == lp);
    sc_top -= n;
    nspill = lp - sc_top - SC_SIZE;
    for (i=0; i < nspill; ++i) {
        if (m_top > lp) {
            mem[m_top] = sc[m_top & SC_MASK];
        }
        --m_top;
    }
    if (lp_pulldown) {lp -= sc_top;}
    else if (lp > m_top) {lp = m_top;}
}
```

Figure 4.2: The `sres` instruction in presence of `LP`. The number of spilled words is reduced since the `LP` is less than `m_top`.

mented in case it is below the `sc_top`. No further action is required. Figure 4.3 shows the pseudo code of the `sfree` instruction.

```
void free(int n) {
    sc_top += n;
    if (sc_top > lp) {
        lp = sc_top
    }
    if (sc_top > m_top) {
        m_top = sc_top;
    }
}
```

Figure 4.3: The `sfree` instruction drops `n` elements from the stack cache. This instruction may change the `LP` pointer according to the `sc_top` pointer.

`sens` n**:**
The stack cache's `sens` instruction does not modify the `sc_top` and may only increase the `m_top`. The invariant is, thus, trivially respected. Moreover, coherency of the data loaded into the cache is guaranteed. The `sens` instruction thus requires no modification. Figure 4.4 shows the pseudo code of the `sens`

```
void ensure(int n) {
    int nfill, i;
    nfill = n - (m_top - sc_top);
    for (i=0; i < nfill; ++i) {
        ++m_top;
        sc[m_top & SC_MASK] = mem[m_top];
    }
}
```

Figure 4.4: Pseudo code of the `sens` instruction. Lazy spilling does not modify the `sens` instruction.

instruction.

`store:` The stack store instruction writes to the stack cache and may thus change the value of data in the stack cache that is coherent with the main memory before the store instruction writes to the stack cache. Therefore, the `LP` needs an adjustment whenever the effective address of the store is larger than the `LP`, i.e., the `LP` needs to be greater than the effective address of the store instruction. Figure 4.5 shows the pseudo code of the store instruction.

```
LP_new = Max(LP_old, effective_address of store)
```

Figure 4.5: The `store` instruction effect on lazy pointer value.

In the following we make the lazy spilling mechanism clear with a detailed example.

Example 4.1 *Assuming again the program from Figure 4.1, the first time that function* `bar` *is executed, the stack cache is empty. All the three pointers (*`sc_top`*,* `m_top` *and* `LP`*) point to the top of the stack at address* 256*. As before, the* `sres` *instruction moves the* `sc_top` *down to address* 248*. Since* `sc_top` = `LP` *and we assume that the newly reserved and uninitialized space is coherent with the main memory, the* `LP` *moves along with the* `sc_top`*. Next, the store instruction (l. 4) modifies some data present in the stack cache. Some content of the stack cache is thus no longer consistent with the main memory. The* `LP` *consequently has to move upwards and now points to address* 252*. Then, the first call to the function* `foo` *causes two words to be spilled (see 4.1.1) and the* `m_top` *is thus decremented to* 248*. As the* `LP` *is larger than the* `m_top` *at this point, the* `LP` *would normally require an adjustment. However, since we assume*

that `foo` *reserves the entire stack cache, it is safe to set the* `LP` *to the value of* `sc_top` *(*216*). Any stores within function* `foo` *then automatically adjusts the* `LP` *as needed. After returning from function* `foo`*, the stack cache is again empty. All three pointers of the stack cache then point to address* 248*. Then, the* `sens` *reloads the stack frame of the function* `bar` *(l. 11), leaving both* `sc_top` *and* `LP` *unmodified. However, the* `m_top` *increments to* 256*. The occupancy of the normal stack cache at this point is* 8 *(*2 *words), while the effective occupancy of the stack cache with lazy spilling is* 0*, i.e., the entire cache content is known to be coherent with the main memory. The effective occupancy for the next call to* `foo` *as well as of the ones of the subsequent iterations of the loop, stays at* 0*. The reserve within* `foo` *thus finds the entire stack cache content coherent with the main memory and avoids spilling completely. Finally, when the program exits the loop, the* `sfree` *instruction (l. 16) increments the* `sc_top` *to* 256*. Moreover, to satisfy Eq. 4.2,* `LP` *has to be incremented. Note that for the standard stack cache and the lazy spilling stack cache, the* `sc_top` *and* `m_top` *values stay the same in all the steps above.*

In Chapter 6 we provide the effect of the lazy spilling method on improving the performance of the original stack cache.

In the previous Chapter 3, we explained that the standard stack cache is implemented by a controller, which executes spill and fill requests, and control instructions (`sres`, `sfree`, `sens`) sending requests to that controller. Thus, the controller is independent from the extensions presented for the lazy spilling. Therefore, we can simply extend the `sres`, `sfree`, and stack store instruction as indicated in Section 4.1.4. The stack cache reads the `sc_top` and the `m_top` in the decode stage. Likewise, the `LP` is read in the decode stage as well, which allows us to perform all `LP`-related updates in the decode and execute stages. This way, additional logic on the critical path in the memory stage, where the cache is actually accessed, is avoided. This applies in particular to the store instruction, whose effective address only becomes available in the execute stage. For lazy spilling, a single additional register is needed (`LP`). Updating the `LP` in the `sres` instruction adds two multiplexers to the original implementation. The `sfree` and stack store instructions each need an additional multiplexer.

4.1.5 WCET analysis

As we explained, the filling behavior of the ensure instructions is not affected by lazy spilling, therefore, the WCET analysis, related to this instruction, remains unchanged in compared to the original analysis. Without lazy spilling, in the reserve instruction, the stack cache *occupancy*, i.e., the utilized stack cache

region of size `m_top` − `sc_top` is a determining factor for the WCET analysis. However, in a lazy spilling stack cache, to benefit from the reduced overhead, the possibly smaller region `LP` − `sc_top` should be considered. Further, details of the WCET for the lazy spilling stack cache are out of the scope of this work.

In the next section, we propose and describe another method for improving the performance of the stack cache.

4.2 Block-aligned stack cache

As we mentioned in Chapter 3, the memory transfers generated by the standard stack cache are not generally aligned. These unaligned accesses risk introducing complexity to the otherwise simple WCET analysis and leads to less optimal utilization of the main memory's bandwidth. In this section, we investigate three different approaches to handle the alignment problem in the stack cache:

- Unaligned transfers.
- Alignment through compiler-generated padding.
- A novel hardware extension ensuring the alignment of all transfers.

4.2.1 Motivation

As we explained in the previous Chapter 3, the stack cache uses three instructions to reserve space on the stack (`sres`), free space (`sfree`), and to ensure the availability of stack data in the cache (`sens`). The reserve and ensure instructions may cause memory transfers between the stack cache and the main memory and thus are impacting the WCET. One characteristic of these transfers is that the start addresses of them are not guaranteed to be aligned to the memory controller's burst size.

To illustrate this problem, we consider the function calls shown in Figure 4.6 and a stack cache of size 8 words, i.e. address 256 is the top of the stack. Moreover, we assume that the burst size of the main memory's controller is 4 words and that on the start of the program no stack data has been allocated on the stack cache and it is entirely empty. Thus, the `m_top` and `sc_top` pointers are both pointing to the top of the empty stack at address 256. Therefore, any transfer between the stack cache and the main memory should happen at addresses that are multiples of 4 words, e.g. 256, 240 and so on.

Function `bar` uses the stack cache to allocate a stack frame of two words (l. 2) and function `foo` uses the stack cache to allocate a stack frame of seven words (l. 10). Reserving seven words on the stack cache causes spilling one word to the main memory from the top of the stack, i.e. address 256. Once function `foobar` is called, reserving six words on the stack cache, the spilling starts from the address 252, that is not a multiple of 4 words. Therefore, the stack cache has to start the transfer with invalid data from an aligned address. As we can see from the example, the unaligned transfers incur a performance penalty. Misaligned accesses may require up to three additional words to be transferred. Thus, a misaligned transfer that has a nominal smaller size and is not the worst-case in terms of transfer cycles, might actually take longer than the aligned with larger sizes. This is in contrast to traditional caches where the cache line size is typically aligned with the burst size. Thus, when the data is not aligned, with regard to the average performance, this is less of an issue and may only lead to a less optimal utilization of the main memory's bandwidth.

For the WCET the issue is more problematic and for a precise WCET analysis we should take this penalty into account. Static analysis [26, 97] of caches typically proceeds in two phases: (1) potential addresses of memory accesses are determined, (2) the potential cache content for every program point is computed. As we mentioned for standard caches, the alignment usually is not an issue, as the size of the cache can be aligned with the main memory's burst size. The design of the stack cache however, eliminates the need for precise knowledge of addresses, thus eliminating a source of complexity and impreci-

```
1  function bar()
2  sres 2
3  call foo
4  // reload local stack frame
5  sens 2
6  // exit function
7  sfree 2
8  ret
9  function foo()
10 sres 7
11 call foobar
12 // reload local stack frame
13 sens 7
14 // exit function
15 sfree 7
16 ret
17 function foobar()
18 sres 6
19 add ...
20 // exit function
21 sfree 6
22 ret
```

Figure 4.6: An example for three nested function calls, allocating data on the stack cache, causing unaligned memory transfers.

sion in analysis. However, the unaligned transfers introduce complexity to the otherwise simple WCET analysis of the stack cache. The most obvious solution for WCET analysis regarding the unaligned transfers is to assume that we need to transfer additional blocks for all the transfers, either aligned or non-aligned. However, this is a rather pessimistic assumption. In particular, for cases when we have smaller sizes of transfers this assumption leads to even more pessimistic WCET estimations. Therefore, the alignment of the stack cache content needs to be known or otherwise all the access have to be assumed unaligned. The alignment of stack cache content is highly dependent on the execution history and therefore increases the analysis complexity. Another option is to define a new analysis problem that takes alignment into account. The alignment of the transfers depends on the history of previous transfers. Thus, we need to capture the change to alignment explicitly for each potential transfer that happened before a specific `sres` or `sens` instruction. Therefore, for improving the WCET and average performance of the stack cache we propose the block-aligned stack cache.

Here, we look at three methods to handle the alignment problem for the stack cache:

- A stack cache initiating unaligned transfer
- Compiler-generated padding to align all stack cache allocations (and thus all transfers)
- Our novel hardware extension that guarantees that all stack cache transfers are block-aligned.

Our novel method offers a simple solution with low hardware overhead, no extra pessimism, no extra complexity and good average-case performance. For the hardware extension a burst-sized block of the stack cache is used as an *alignment* buffer. This block makes the block-aligned memory transfers possible. The downside of this approach is that the effective stack cache size is reduced by one block. On the other hand, the block-aligned transfers simplify WCET analysis. In addition, this method allows us to perform all the stack cache operations at word granularity, which improves the cache's utilization. Moreover, for the WCET analysis, we eliminate the need to track the alignment of the stack cache content. The hardware overhead of our approach is minimal: the implementation of `sres` and `sens` is simplified, while `sfree` requires some minor extensions. The free instructions may need to initiate memory transfers to preserve the alignment of the stack cache content. In the following we explain the detailed implementation of block-aligned stack cache instructions.

4.2.2 Implementation

This section explains the hardware extension to the original stack cache that guarantees that the stack cache only initiates aligned memory transfers, i.e., the start address as well as the length of the memory transfer are multiples of the memory's alignment requirement (the burst size).

The stack cache is organized in blocks matching the burst size. Moreover, we logically reserve a block in the stack cache as an alignment buffer. Note that this reserved block is not fixed, instead the last block pointed to by `m_top` *dynamically* serves as this alignment buffer. This block is not accessible, for instance, to the compiler, and thus reduces the effective size of the stack cache by one block. The buffer allows us to align all the memory transfers to the desired block size. With regard to the original stack cache, this corresponds to an additional invariant that needs to be respected by the stack cache hardware. Considering the size of the block to be `BS` we need to make sure the following invariant always holds:

$$\texttt{m\_top} \mod \texttt{BS} = 0 \tag{4.3}$$

To respect this new invariant the stack control instructions have to be adapted as follows:

`sres` n**:**
Subtract n from `sc_top`. If the occupancy exceeds the stack cache size, a *spill* is initiated, which lowers `m_top` by multiples of `BS` until the occupancy is smaller than $|SC|$. Figure 4.7 shows the pseudo code of the `sres` instruction. We use the same constants from previous section in the pseudo codes: the stack cache size is assumed to be `SC_SIZE` and the mask, `SC_MASK`, equals to `SC_SIZE`-1.

As we explained before, the effective occupancy of the stack cache is reduced by the `BS` value. Therefore, the argument value all for all the block-aligned stack cache's instructions should respect the following invariant:

$$n \leq |SC| - \texttt{BS} \tag{4.4}$$

We specifically mention this invariant before the start of the pseudo code for all the three instructions to emphasis that the compiler should not allocate more

```
// requires: n <= SC_SIZE - BS
void reserve(int n){
    sc_top -= n;
    while (m_top - sc_top &gt; SC_SIZE){
        // burst of BS:
        sc[(m_top - BS) & SC_MASK] --> mem[m_top - BS];
        m_top -= BS;
    }
}
```

Figure 4.7: Pseudo code of the `sres` instruction for the block-aligned stack cache.

words on the block-aligned stack cache than specified by the invariant 4.4.

`sens` n**:**
If the occupancy is not larger than n, a *fill* is initiated, which increments `m_top` by multiples of `BS` so that $\texttt{m\_top} - \texttt{sc\_top} \geq n$. Figure 4.8 shows the pseudo code of the `sens` instruction.

```
// requires: n <= SC_SIZE - BS
void ensure(int n) {
    while (m_top - sc_top < n) {
        m_top += BS;
        // burst of BS:
        sc[(m_top - BS) & SC_MASK] <-- mem[m_top - BS];
    }
}
```

Figure 4.8: Pseudo code of the `sens` instruction for the block-aligned stack cache.

It is easy to see that the modifications to the `sres` and `sens` are minimal. Clearly, when Eq. 4.3 holds, spilling and filling in multiples of `BS` ensures that the equation also holds after these instructions. In addition, the reserved block serving as an alignment buffer guarantees that sufficient space is available during filling to receive data and sufficient data is available during spilling to transmit data.

`sfree` n:
Add n to `sc_top`. If `m_top` < `sc_top`, set `m_top` to the smallest multiple of `BS` larger than `sc_top` and fill a single block from main memory. Figure 4.9 shows the pseudo code of the `sfree` instruction.

```
//requires: n <= SC_SIZE - BS
void free(int n) {
    sc_top += n;
    if (sc_top > m_top) {
        m_top = (sc_top + BS - 1) & ~(BS - 1);
        if (m_top != sc_top) {
            // burst of BS:
            sc[(m_top - BS) & SC_MASK] <-- mem[m_top - BS];
        }
    }
}
```

Figure 4.9: Pseudo code of the `sfree` instruction for the block-aligned stack cache.

As we can see, changes are more complex for `sfree`. Whenever a number of `sfree` instructions are executed in a sequence such that the occupancy becomes zero, the `m_top` pointer needs to be updated. In order to satisfy Eq. 4.3, two options exist: (a) set `m_top` to the largest multiple of `BS` *smaller* than `sc_top` or (b) set `m_top` to the smallest multiple of `BS` *larger* that `sc_top`. The former option would mean that the cache's occupancy becomes negative, which would entail non-trivial modifications to the other stack control instructions. The second option, which represents a non-negative occupancy, thus is preferable. However, in order to guarantee that the content of the stack cache reflects the occupancy equal to the `m_top` − `sc_top` value, a single block has to be filled from the main memory.

Figure 4.10 shows the content of the logical stack during four nested function calls using three different stack cache configurations, i.e. the unaligned, the padded and block-aligned stack cache with a stack size of 128 bytes (we consider bytes instead of words for simplicity of example) and 32 byte blocks. The stack frames of these functions would require 61, 24, 34 and 54 bytes of stack space (respective to the order of calls, i.e. A, B, C and D). In the unaligned stack cache 4.10a, after the first three function calls, 119 bytes in the stack cache are occupied. When the last function (i.e. D) is called, the stack cache spills 45 bytes to free up the required bytes for this function to unaligned address. In the block-aligned stack cache 4.10b, after the first three function calls, like the previous case, 119 bytes in the stack cache are occupied. The stack cache spills

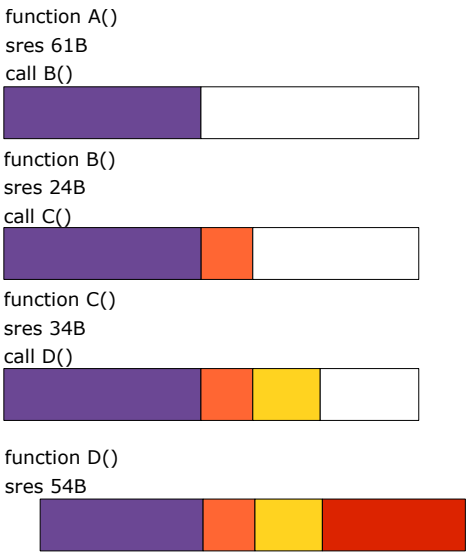
64 bytes to an aligned address to free up space for the fourth function. In the padded spilling 4.10c, the stack frame sizes are aligned to 32, giving 64, 32, 64 and 64 bytes for the four functions (respective to the order of calls). After the first two function calls 96 bytes are occupied, thus, the stack cache spills 32 bytes to an aligned address for the third function and when fourth function is called, another 64 bytes are spilled again.

4.2.3 WCET analysis

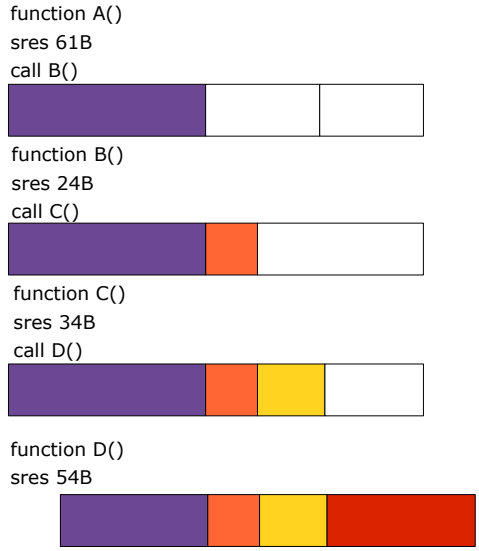
Since the only change to the `sres` and `sens` instructions is that the spilling and filling are done in multiples of `BS`, analysis of the original stack cache is applicable to the block-aligned stack cache as well. However, the main difference is that the timing of `sfree` instructions also has to be analyzed. The `sfree` instruction requires performing a fill when necessary and therefore the `sens` analysis can be used for the `sfree` instruction as well. More details of the analysis are out of the scope of this work. In Chapter 6 we provide the evaluation of the block-aligned stack cache.

4.3 Virtual stack caching

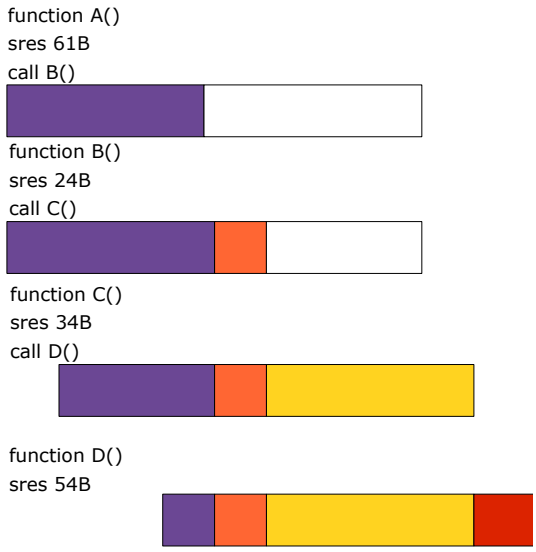
We explained the stack cache's simple structure in previous chapter. However, this simple structure of the stack cache has drawbacks. One problem is that when multiple tasks are executed using preemptive scheduling, the two pointers only capture the cache state of the task currently running. Thus, the states of other (preempted) tasks are *lost* because `sc_top` and `m_top` get overwritten, even if the data of preempted tasks might still be in the cache. The stack cache's hardware *cannot* ensure that this data remains unmodified. Another problem is that it cannot ensure that modified cache data, not yet written back to main memory, remains coherent. As a consequence, the entire stack cache content has to be *saved* to the main memory when a task is preempted. Saving the entire stack cache imposes the *restoring* stack cache content before that task is resumed. Saving and restoring the content may induce considerable overhead that has to be accounted for during the analysis of a real-time system equipped with a stack cache.



(a) Unaligned spilling.



(b) Block-aligned spilling.



(c) Padded spilling.

Figure 4.10: Logical stack during function calls with different methods of stack cache alignment, (a) unaligned spilling (b) block-aligned spilling and (c) padded spilling.

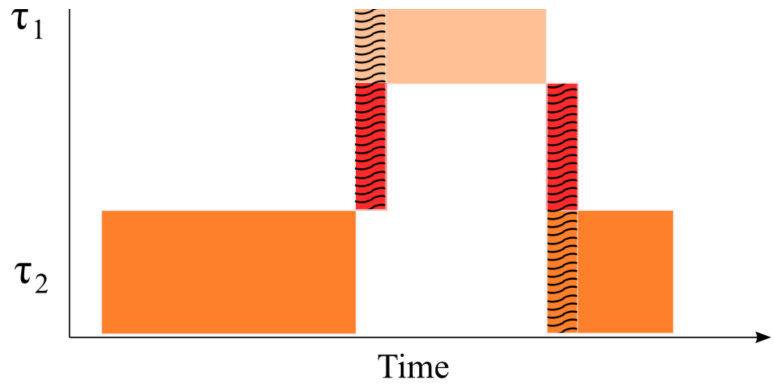


Figure 4.11: Context switch overhead caused by preemption.

4.3.1 Schedulability analysis issue

When a preemption occurs, the content of classical data caches will be updated as the preempting task perform memory accesses, i.e. when misses occur. When the preempted task is resumed, an additional Cache- Related Preemption Delay (CRPD) must be accounted for computing the WCET due to data blocks that were evicted by the preempting task. A response time analysis integrating CRPDs can then be performed, as in [6] for instance. When considering a stack cache, the stack data of the preempted task must be saved before the stack cache of the preempting task can be restored. Only then, the preempting task can start execution. A stack cache's analysis can determine the bounds for restoration costs, which have to be added to the preempted task's WCET similar to standard CRPDs. However, the costs determined by analysis have an immediate impact on the preempting task (similar to effects of a standard cache with a write-back policy [100]). Let us illustrate this through an example.

Example 4.2 *Figure 4.11 shows a high-priority task τ_1 preempting a low-priority task τ_2. Before τ_1 can start execution, the content of τ_2's stack cache is saved to main memory. Thus, τ_1 has to wait until the memory transfer of the low-priority task (represented by the red block on the left in Figure 4.11) is completed. When τ_1 finishes, τ_2's stack cache content is brought back from main memory (represented by red block on the right) causing another delay when τ_2 is resumed. τ_1 therefore suffers from an additional delay that depends on the amount of data of τ_2 to be transferred.*

Apart from an increased Worst-Case Response Time (WCRT) of high-priority tasks, this delay can also vary heavily and cause undesirable jitter, depending on the preempted tasks and their respective context saving delays. A similar issue exists in caches with write-back policies, which are not recommended for real-time systems [100]. While the stack cache simplifies the WCET analysis of a single task, this additional CRPD, that depends on the preempted tasks, complicates the WCRT analysis when using preemptive schedulers.

4.3.2 Virtual stack cache design

To mitigate this problem, we propose to allocate a Virtual Stack Cache (VSC) to each task, i.e. each task has its own dedicated VSC. These caches are then mapped to a fast local scratchpad memory, shared among all these tasks (i.e., running on the same physical core). For now, let us assume that all the VSCs of a system fit into the underlying memory. The context saving and restoration costs are then completely eliminated. It suffices to retrieve the location where the VSC of the preempting task is mapped, which, in the simplest case, means fetching two pointers. The scheduling issue pointed out above, simply disappears along with the preemption overhead. The hardware, naturally, has to keep track of the VSC locations at the processor-level given by an offset (`vscOffset`) and size (`vscSize`) pointer. Each task's stack area is then located in the range [`vscOffset`, `vscOffset` + `vscSize`] in the underlying memory. On a context switch, only the `vscOffset` and `vscSize` pointers have to be stored, while no transfer of stack data is required. Scratchpad memories are typically small and expensive, which limits the number of VSCs that can be stored simultaneously under a statical partitioning. It also appears to be a waste of resources to keep inactive stack data in the scratchpad. Clearly, a more efficient solution is needed that allows to off-load VSCs to off-chip memory when the stack data is not needed. VSCs lend themselves for such a (semi-)dynamic scheme, since their mapping can freely be updated (even more, the size of VSCs could be updated dynamically). We thus envision that VSCs are combined with an arbitration mechanism that allows the system's task scheduler to dynamically save and restore the VSCs of inactive tasks to/from main memory. Since the task scheduler is controlling this process, we believe that a time-predictable solution is feasible, which as a side effect improves the system's resource utilization.

4.3.3 Scheduling opportunities

Managing the stack cache using a dynamic partitioning strategy clearly requires implementing prefetching techniques, for memory transfers to/from the stack

caches, based on future decisions taken by the underlying scheduler. A timing constraint exists for these memories transfers: they must be completed before the tasks are executed so that load and store accesses to the stack cache are guaranteed hits. To illustrate this, let us assume a static scheduling and that at most two VSCs can be mapped in the stack cache. While some task τ_i is running, the VSC of another task τ_j could be filled, so that when τ_j is scheduled, its stack data is available. Then, while τ_j is running, the VSC of τ_i can be spilled before the stack cache of the next task to be scheduled is filled. Using this approach, the CRPD issues for the stack cache can be avoided as memory transfers are performed in the background, while the processor is running a task. Context saving and restoring analyses can be used to bound the time needed to perform memory transfers and ensure that VSCs are available before tasks are executed. The analysis results can furthermore be used to determine those program points where preemption overhead is the smallest.

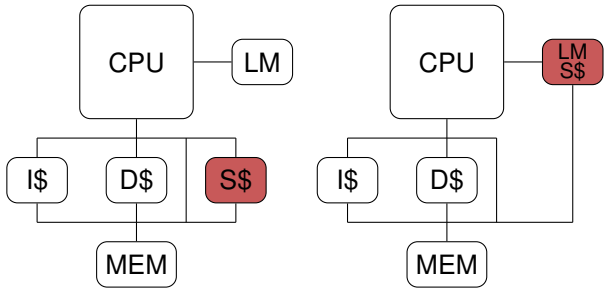
This approach depends on the memory bandwidth that is available for performing such memory transfers. In a single-core system this usually depends on the number of free bus slots. Multi-core systems, on the other hand, often use Time Division Multiplexing (TDM), where each processor accesses main memory during a dedicated time slot (via a network or bus). VSCs then can be transferred to/from main memory in unused TDM slots. Section 6.4.1 presents an evaluation of the average-case memory bandwidth that is available when using a TDM approach. However, we leave a formal timing analysis and its integration within extensions to the task model for the schedulability analysis as future work. Note that a dynamic scheduling can be used in conjunction with a dynamically partitioned stack cache. However, this would probably result in wasting memory bandwidth. Indeed, each time the scheduling structures are updated, a different task τ_j than the one whose VSC is currently being prefetched (τ_i) may be identified as being the next task to be executed. Thus, the VSC of τ_j must be transferred, making the previously speculatively prefetched stack cache of τ_i useless. The challenge is then to provide off-line guarantees for the memory transfers related to the stack caches. Finally, if the memory transfers associated to the VSC of a task cannot be guaranteed to be finished before the task activation, the order of memory transfers could be optimized to start with data that is first used by the task. However, this requires a model of patterns of memory accesses performed by the tasks. The bet is that sufficient time intervals are available to finish memory transfers of data before they are actually used by the tasks.

4.3.4 Optimized stack caches partitioning

We now discuss some optimizations related to the partitioning of the virtual stack caches. When more than two VSCs can be mapped in the memory underlying the stack cache, it would be interesting to study how to allocate, in the time and space domains, the VSCs in order to minimize their memory bandwidth consumption. For instance, the VSC of most often executed task could be statically allocated, while dynamically allocating the VSC of other less frequent tasks. It would also be interesting to tune the partitioning of the stack cache based on the priority levels of tasks, e.g., in the presence of permanent hardware faults. For instance, when a fault is detected in a memory block holding a VSC of a given task τ_i, three options would be possible. First, the content of the VSC of τ_i could be remapped to a different location of the memory. If not possible, the second option is then to reduce the size of the VSC of tasks whose priorities are lower than τ_i. Finally, the last option is to remove one or several VSCs of tasks whose priority is lower than τ_i. This may have an impact on the WCET of these lower priority tasks depending on whether stack cache filling/spilling can still be hidden or not. However, the same performance is guaranteed for the higher priority tasks.

4.3.5 Hardware implementation

The virtual stack cache extension, which aims at partially hiding the preemption cost associated with the stack cache, is also evaluated within the Patmos platform. For this we adapted the platform's hardware model by introducing two new special registers, by modifying the processor's bus connections, and by updating the 5-stage processor pipeline. Figure 4.12a shows Patmos' original memory subsystem, integrating a single stack cache (S$), an instruction cache (I$), a data cache (D$) and a local scratchpad memory (LM). Separate buses are used to communicate between the processor core (CPU) and the caches on one side and the local memory on the other side. All caches are connected to a shared global memory (MEM), which is arbitrated using TDM in case of a multi-core platform. The Patmos processor has a local memory with constant access time the same as the stack cache's access time. Hence, we use this property to merge the stack cache with the local memory. Thus, to implement virtual stack caches, we propose to restructure the bus infrastructure, and to merge the stack cache's memory with the existing local memory. Merging the stack cache with the local memory gives us the flexibility to move and resize the stack caches (that are belonging to different tasks) within the local memory (as we discussed earlier). For this, we need to remove the stack cache from the bus between the CPU and the data/instruction caches 4.12b. However, the bus



(a) (b)

Figure 4.12: Block diagram of the memory subsystem components of a processor, (a) using a normal stack cache, (b) using a stack cache merged with a scratchpad memory.

between the caches and the global memory remains untouched. By default, the stack cache registers (`sc_top`, `m_top`) are read in the third stage (`EX`). Therefore, we also read the two new `vscOffset` and `vscSize` registers in the `EX` stage. In addition, the address calculation of stack-cache-related loads and stores has to be updated. The computations are done in the 4th stage (`MW`) within the processor. Accesses to a virtual stack cache thus do not differ from regular accesses to the local memory.

4.4 Summary

In this chapter we introduced two different methods to improve the stack cache's performance, lazy spilling and block alignment. The lazy spilling method is an optimization of the standard stack cache to avoid redundant spilling of the cache content to main memory, if the content is coherent with the main memory. The lazy spilling can be analyzed with little extra effort compared to the original stack cache, and thus, benefits the worst case spilling behavior of the stack cache that is important for a real-time system. Compiler generated padding is a simple solution to the alignment problem for the stack cache, However, it generally leads to increased spilling and filling as well as a reduced utilization of the stack cache. Generating unaligned memory transfers naturally performs well for the average case. Although, unaligned memory transfers complicate WCET analysis since the alignment of the stack data. The solution proposed in this chapter, the block-aligned stack cache, offers a reasonable trade-off, which

combines moderate hardware overhead with good average-case performance.

Finally, we proposed to *virtualize* the stack cache. Several virtual caches of different tasks can then be stored in a large local scratchpad memory, which allows us to quickly switch from one virtual cache to another. The virtual caches of preempted tasks can, furthermore, be saved/restored in parallel with the execution of another task by using unused TDM/bus slots to access the off-chip main memory. Measurements indicate that typical programs easily allow the transfer of several stack caches. We described the design and implementation of the proposed methods. Moreover, we have integrated our designs with the time-predictable processor Patmos. We present the evaluation of the three proposed methods in details in Chapter 6.

Chapter 5

A Software Managed and a Hardware Managed Stack Cache

This chapter looks at stack caching from two different angles: software managed and hardware managed stack caching. We first present the design and implementation of software managed cache for stack allocated data. We present the compiler-aided implementation of this cache within the LLVM compiler framework. Secondly, we propose caching the stack data for standard reduced instruction set computing (RISC) processors, the hardware managed stack cache. The hardware managed stack cache requires no changes in the ISA and compiler. We describe our implementation in hardware, transparent to the ISA and compiler. Moreover, we explore the effect of the stack cache type on its performance.

This chapter is structured as follows: in Section 5.1 we describe the idea of a software managed stack cache. We describe the detailed implementation in software and porting the implementation to the Patmos processor. In section 5.2 we provide details on the implementation of the hardware managed stack cache and effect of the stack cache type on the average performance. We finish the chapter by highlighting the main ideas and contributions of this section in the summary Section 5.3.

5.1 Software managed stack cache

While a hardware managed stack cache requires modifications to the instruction set to implement the stack cache semantics, For cases where changing the hardware is not an option, we propose to implement the stack cache in software and assisted by the compiler.

Implementing the software managed stack cache requires both of the implementation of the stack cache specific functions and compiler support. In the following, we explain the details of the stack cache functions implementation as well as the changes required to adapt the compiler supporting these functions. We explain the details of the stack cache functions implementations in the following. The compiler changes for supporting the software managed stack cache are out of the scope of this thesis.

5.1.1 Stack cache functions

The software managed stack cache is a scratchpad memory organized as a ring buffer. For manipulating this ring buffer, we define two pointers: *stack top* (`sc_top`) and *memory top* (`m_top`). The `sc_top` is the address of the top of the stack data and the `m_top` is the address of the top element in the main memory. Both pointers point into the stack area in the main memory. To redirect stack accesses to the SPM, the addresses are *translated* to point into the address area where the SPM is mapped. The SPM can also be seen as a sliding window into the main memory.

We initialize the `sc_top` and `m_top` to the same value. The difference between `sc_top` and `m_top` is the occupied space (fill level) of the stack cached in the SPM.

Two registers are used for the stack cache: (1) the stack pointer `sc_top` pointing to the top of the stack, as index into the main memory and (2) the memory pointer `sc_mem` pointing to the last spilled word in main memory. At program start both pointers point to the same address, the word above the memory area reserved for the stack. The difference between these two pointers is the number of words from the stack that is cached in the SPM. Since our approach relies completely on software and compiler support, we cannot use the special registers of the Patmos processor for the two stack cache pointers, as we did in the previous chapters. Hence, we use two general purpose registers and the compiler directly reserve these two registers for the `sc_top` and `m_top` pointers (we chose registers $r27$ and $r28$, however, any other register that is not reserved

```
asm ( assembler template
: output operands /* optional */
: input operands /* optional */
: list of clobbered registers /* optional */);
```

Figure 5.1: Accessing the internal registers of the processor using the inline assembly [29].

for any other special purpose (such as the return address register) can be used as well.

To access these registers and read and write them, we use the extended inline assembly [29]. With the extended asm we can read and write C variables from assembler, using the format in Figure 5.1. Moreover, it allows us to specify the input registers and output registers. As we clobber the $r27$ and $r28$ hardware registers, we have to list these two registers in the clobber-list. Defining these registers as clobbered informs the compiler that we use and modify these registers ourselves and the compiler will not assume that the values that it loads to these registers are valid.

The template in Figure 5.1 consists of assembly instructions. Colons separates the assembler template from the input and output operands. Operands are separated by colons as well. It should be noted that we use the *volatile* asm statement to make sure the compiler executes it where we put it, i.e. the compiler cannot movie it for optimization purposes.

It should be noted that for accessing the processor's scratchpad memory and main memory, we used the *_SPM* and *_UNCACHED* macros to declare data structures pointing to these memories. Using the *_UNCACHED* macro, the compiler maps the declared memory to the main memory and bypasses the data cache. The *_UNCACHED* macro, maps the memory to the scratchpad address space.

In the following we show the implementation of the three stack manipulation functions, `reserve`, `free`, and `ensure`.

`reserve` x**:**
The `reserve` function is shown in Figure 5.2. This function is called on function entry to reserve the stack frame for the function. It reserves `n` words in the SPM for the called function by decrementing the stack pointer by `n`. If the new stack frame overlaps with older stack content in the SPM, that older stack

```
void _sc_reserve()
{
  int n, m_top, sc_top;
  unsigned spilled_word;
  int i;
  int n_spill;
  asm volatile("mov %0 = $r1;"
  "mov %1 = $r27;"
  "mov %2 = $r28;"
  : "=r" (n), "=r"(sc_top), "=r"(m_top)
  ::);
  sc_top -= n * 4;
  n_spill = (m_top - sc_top - (int) SPM_SIZE) / 4;
  for (i = 0; i < n_spill; i++){
    m_top -= 4;
    _SPM unsigned *spm = (_SPM unsigned *) (m_top & MASK);
    _UNCACHED unsigned *ext_mem = (_UNCACHED unsigned *) (m_top);
    spilled_word = *spm;
    *ext_mem = spilled_word;
  }
  asm volatile("mov $r27 = %0;" // copy sc_top to st
  "mov $r28 = %1;" // copy m_top to ss
  :
  : "r"(sc_top), "r"(m_top)
  : "$r27", "$r28");
}
```

Figure 5.2: The `reserve` function reserves `n` free words in the stack cache. It may spill data into main memory.

data is spilled to the main memory. As we see, before the start of the `reserve` operation, we first read the values of the two stack pointer registers into local variables and when the `reserve` operation finishes we store these variables back to $r27$ and $r28$ registers.

`free` x:
The `free` function, as shown in Figure 5.3, frees the stack frame of a function. It is called before function return. No data is exchanged between the SPM and main memory. Only the stack pointer `sc_top` is incremented and if there is no more data cached in the SPM, `m_top` is set to `sc_top`. Like the `reserve` function, we have to read the registers and the operand of the function, before the start of the operation, into local variables and write these variables back to registers once the `free` operation is completed.

```
void _sc_free(){
  int sc_top, m_top, n;
  asm volatile("mov %0 = $r8;" // copy argument to n
  "mov %1 = $r27;" // copy st to sc_top
  "mov %2 = $r28;" // copy ss to m_top
  : "=r" (n), "=r"(sc_top), "=r"(m_top) /* output regs */
  ::);
  sc_top += n*4;
  if (sc_top &gt; m_top) {
    m_top = sc_top ;
  }
  asm volatile("mov $r27 = %0;" // copy sc_top to st
  "mov $r28 = %1;" // copy m_top to ss
  : /* no output regs */
  : "r"(sc_top), "r"(m_top) /* input regs */
  : "$r27", "$r28" /* clobbered */);
}
```

Figure 5.3: The `free` function drops `n` elements from the stack cache. It may change the top memory pointer `m_top`.

`ensure` x:
The `ensure` function, as shown in Figure 5.4, ensures that the actual stack frame is resident in the SPM. It is called by the caller after function return from the callee. The callers stack frame is `n` words large. If the content of the stack in the SPM is less than `n` words, the missing words are filled from main memory. As the two previous functions, we read the registers and the operand of the function into local variables before the operation and write these variables back once the `ensure` operation is completed.

Load and Store x:
Load and store instructions accessing the stack are with a displacement `disp` relative to the stack pointer `sc_top`. As the stack data is always in the SPM, those loads and stores are single cycle latency, similar to a hit in a data cache. The implementation of stack load and store functions is shown in Figure 5.5. The load and store addresses need a translation from the stack area in the main memory to the address area where the SPM is mapped. In practice the size of the SPM (or the part that is used for stack caching in the SPM) is a power of 2. In that case the modulo operation is reduced to a simple bit masking *and* instruction.

```
void _sc_ensure(){
  int n, m_top, sc_top;
  unsigned filled_word;
  int i, n_fill;
  asm volatile("mov %0 = $r8;" // copy argument to n
  "mov %1 = $r27;" // copy st to sc_top
  "mov %2 = $r28;" // copy ss to m_top
  : "=r" (n), "=r"(sc_top), "=r"(m_top)
  ::);
  n_fill = (n*4 - (m_top - sc_top)) / 4;
    for (i = 0; i < n_fill; i++){
    _SPM unsigned *spm = (_SPM unsigned *) (m_top & MASK);
    _UNCACHED unsigned *ext_mem = (_UNCACHED unsigned *) (m_top);
    filled_word = *ext_mem;
    m_top += 4;
  }
  asm volatile("mov $r27 = %0;" // sc_top
  "mov $r28 = %1;" // m_top
  :
  : "r"(sc_top), "r"(m_top)
  : "$r27", "$r28");
}
```

Figure 5.4: The `ensure` function ensures that at least `n` elements are valid in the stack cache. It may spill data from main memory.

```
def load(disp):
return M[SPM_START + ((sc_top+disp) % SPM_SIZE)]

def store(disp, val):
M[SPM_START + ((sc_top+disp) % SPM_SIZE)] = val
```

Figure 5.5: Pseudo code for the stack accessing load and store instructions.

5.1.2 WCET analysis

The motivation for proposing a software stack cache as part of this work is development of computer architectures that are optimized for their WCET instead of the average-case execution time [82]. Therefore, in here we discuss how the software stack cache impacts the WCET analysis.

5.1.2.1 Stack access

Accesses to SPM allocated data use a register as a stack pointer and some address translation. The result of the address translation always points to data within the address range of the SPM. A WCET analysis tool that performs cache analysis can be configured with memory access latencies depending on address ranges. Therefore, as long as static analysis can determine that the effective address of a stack load or store points into the SPM, these instructions will have single cycle latency. Since translated addresses do not escape, a local address (value) analysis is sufficient here. As an alternative, the compiler, which necessarily knows about accesses to the stack cache, can share this information with the analysis tool. This could be achieved by providing per-instruction annotations.

5.1.2.2 Stack spill and fill

The second question is, if the maximum number of fills and spills can be efficiently bounded through static analysis. Without any analysis the conservative bound for each `sres` and `sens` operation would be based on its argument `n` and drastically overestimate the actual (worst-case) behavior. Tracking the worst-case state of the stack cache has been shown by [41] to be an inter-procedural analysis problem. With the stack cache implemented in software and its internal state exposed by the `sc_top` and `m_top` registers, an alternative to a custom-built analysis is to rely on an existing value/loop-bound analysis to analyze worst-case spilling and filling.

The software managed stack cache and the time predictable stack cache proposed in previous chapters, simplify the worst-case execution time (WCET) analysis. However, these methods require changes in the instruction set architecture (ISA) and compiler. Therefore, we propose caching the stack data for standard reduced instruction set computing (RISC) processors. The novelty of this approach relies on the fact that it requires no changes in the ISA and compiler. We add a cache to the hardware, transparent to the ISA and compiler. In the next section we

present the stack cache for RISC processors and describe our implementation in hardware.

5.2 Hardware managed stack cache

Procedure calls and returns in programming languages follow last-in, first-out (LIFO) order. Therefore, the stack structure (with push and pop mechanism) can be used to allocate and deallocate the memory of function calls. Every time a function is called, a unique stack frame is created for it. When programs start executing, an address range in the memory is reserved for the stack data of functions. Accesses to the stack structure are referenced using a stack pointer (sp) register. The address of the access is usually an offset from the current top of the stack address. A $sp + offset$ calculation defines the actual address of the memory access (i.e., load and store instructions). Since the stack data is accessed often, providing quicker access to it can improve the performance.

Moreover, the data cache is known to be a major source of power consumption in processors [95]. Therefore, to reduce the data cache's and hence the whole processor's power consumption, some studies have suggested using the special locality of accesses to the stack data for splitting the accesses to the stack data from the data cache.

Olson et al. [58] propose treating data from the program's stack differently than the non-stack data for reducing dynamic energy. Authors introduce implicit stack cache and explicit stack cache. The implicit stack cache limits the stack data to reside in specific data cache ways. The explicit stack cache uses a separate L1 cache for stack data. Authors show the proposed methods results in significant energy savings (37% and 36% respectively).

Gonzalez et al. [32] propose a stack oriented data cache filtering. The put a small stack filter (4 to 64 words) before the L1 data cache in the memory hierarchy. The stack filter is circular register file and uses two status bits for each register for the valid and dirty status (like standard caches with write back mechanism). To avoid delayed accesses to the L1 data cache, the authors introduce a unit to detect the hit and miss in the stack filter earlier.

Here, we apply the same idea of separating the stack data for improving the performance. We propose, as we mentioned, extending the RISC style processors with a cache specifically storing the stack data for improving the average case execution time. A hardware extension with minimal overhead that requires no compiler support. Figure 5.6 shows separation of the stack data to a dedicated

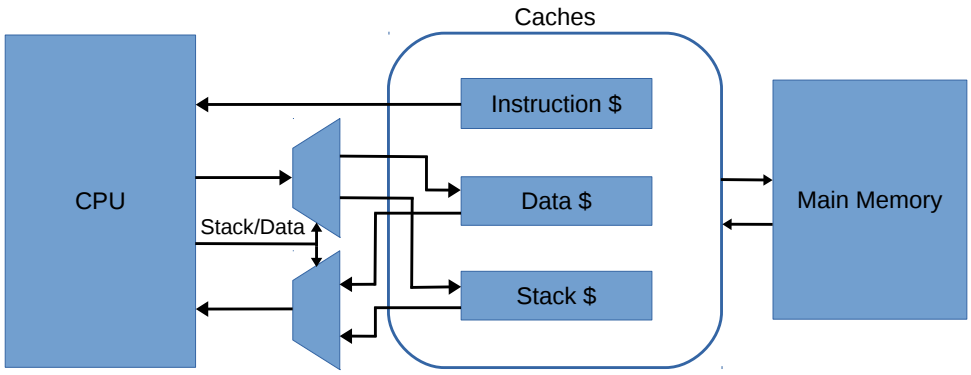


Figure 5.6: Redirecting the stack data to a *stack cache*.

stack cache.

5.2.1 Associativity

To implement this cache, we face the common problem of the hit ratio vs. the search speed. The direct mapped cache is the simplest with best search speed, since there is only one possible place to cache any memory location. To increase the hit ratio of the direct mapped cache we have to increase the number of places in the cache where a block can reside (i.e., increase associativity). A cache is a two-dimensional array of blocks. The rows are sets and the columns are ways. A cache with a single column is a direct mapped cache. In the following, we argue why a cache dedicated to the stack allocated data cannot benefit from the set associativity. As we mentioned, the address of any stack data is relative to the stack pointer. To map the stack data to the stack cache, modulo operation is used. We assume that size of the cache is S. Moreover, we assume size of the stack frame of any given function is smaller than the stack cache size. Any function $F1$, with its stack frame between $sp + offset1$ and $sp+offset2$ maps to addresses between $(sp+offset1)\%S$ and $(sp+offset2)\%S$ in the cache. We can safely assume the $sp\%S$ maps to the first address in the

cache. Considering that all functions frame sizes are smaller than the cache size, the $offset1\%S$ maps to address $offset1$, $offset2\%S$ maps to address $offset2$ and any $offsetN$ between the $offset1$ and $offset2$ maps to a unique address $offsetN$ that is not equal to any other address in this range. Therefore, no two different addresses conflict for accessing a block, and there is no need for associativity. Although, another function ($F2$) might over-write the stack data of $F1$ due to stack cache's size limit, this will not cause a problem. Since the program is already out of the scope of first function and therefore there is no access to the first function's stack data, hence, there will be no conflict misses.

5.2.2 Write through vs. write back

As we explained in Chapter 2, when a write to the cache happens, we can write the data both to the cache and the main memory. This is called write-through policy. Another policy is to write the data only to the cache: the write back policy. Using the write back policy, only when we need to replace a line in the cache that already has been written to (i.e. the *dirty* lines), we transfer the data to the main memory. Hence, the write back caching reduces the transfer cycles to the main memory. However, the write back cache needs to keep track of the dirty lines with dirty bits and a more complicated mechanism for reading and writing and thus has more hardware overhead than write through policy. For example, for a 2K data cache with 4 words cache lines, the write back policy needs 128 dirty bits.

Different addresses belonging to the same cache line, map to the same cache line. On a direct mapped cache, difference between the addresses of different memory blocks that can map to the same cache is greater than or equal to the size of the cache. Therefore, when a function's stack frame size is less than the stack cache size, there will be no conflict misses. Therefore, for a direct mapped stack cache the write back mechanism is the best choice, since it decreases the unnecessary writes to the main memory and reduces the transfer cycles. Whenever a new function over-write an upper function's stack data in the stack cache we can write back the data to the main memory and eliminate the unnecessary write on each write to the stack cache within a function.

5.2.3 Implementation

To implement the proposed idea, we need to identify the stack accesses and direct them to the stack cache memory. There are different methods to do this:

Comparing the source operand of the load/store instruction with the stack pointer register On a typical RISC machine, one general purpose register is used as stack pointer, e.g., MIPS uses register 29 [61]. We can use this register in identifying the stack accesses. Any load/store instruction that uses this register is a stack data access and we can direct it to the stack cache memory (instead of the data cache) [10]. This method can identify the stack accesses in the decode stage before the effective address calculation. Therefore, no extra logic between the memory access and the address calculation is needed. However, not all the accesses to the stack cache are directly using the stack pointer register and can be overlooked just by checking whether they use the stack pointer. Therefore, just relying on the access to the stack pointer register is not enough to classify stack accesses. Therefore, we should find a way to mitigate this problem.

Comparing the address of the load/store instruction with the stack data range To identify all the accesses to the stack data we could use the fact that the addresses of the stack slots in the memory are relative to the current stack pointer. Therefore, we can define a range of addresses for the stack accesses. The stack accesses start from the current stack pointer that can serve as lower bound for stack accesses. We use this method to identify stack data accesses. We should note that generally the compiler does not know the value of the stack pointer and addresses of the stack accesses are known at link time. Hence, the end of the stack is usually implicit. Therefore, we determine an upper bound for the stack data addresses. We define the upper bound for the stack access as: *max_ address*. There are two methods to statically determine the *max_ address*:

- Using a cycle accurate simulator, assuming the *sp* is the stack pointer register, we compare the max_address against the *sp* value on every cycle and update it according to the following:

  ```
  if (max_address &gt sp)
  max_address = sp
  ```

  This method works on a system with a single task running. However, in a system with multiple tasks and a shared memory, the OS can clear the value of the *max_ address* on a task switch. Therefore, we need to use a dedicated register to save the value of *max_ address*, enabling us to save it on a context switch.

- As we mentioned, the value of the symbol pointing to the base of the stack area in the memory is resolved during the linking. Therefore, we can use the symbol information during linking to determine the *max_ address*, i.e., the upper bound for the stack data address.

We can compare the *max_ address* against the effective address of the memory accesses. When the effective address is below the *max_ address* and higher than *sp*, the access is classified as a stack access.

Even though classifying accesses to the memory based on that the stack region size is accurate, any unlikely mis-identification of the stack accesses does not affect the correct execution of the program. Since any misidentified stack access goes to the data cache and gets executed as if the stack cache does not exist. However, since the performance of the system is dependent on the classification of accesses, mis-identification results in less performance.

5.2.4 Cache coherency

As explained we can check the address of the accesses. However, in embedded system the stack as well as the heap can grow and shrink. Therefore, any address might point to the heap data and later in time to the stack data or the other way round. Therefore, in embedded system with real memory addresses we should be sure that stack and heap do not overlap. General purpose processors usually implement virtual memory. Several virtual addresses might be mapped to the same physical address (aliasing). However, caches use physical and not virtual addresses. Therefore, for systems with virtual addressing, cache coherency is not a problem.

Another concern using the physical addresses is that the cache memory might be split to several non-contiguous physical pages and these pages might map to the same block in the stack cache. However, we can safely assume that the size of the stack cache is smaller than normal size of a page and thus the stack cache can fit to one page without causing any mapping problem.

5.2.5 Hardware implementation

To implement the stack caching for RISC processors we chose the Patmos processor [87]. Patmos processor has a five-stage pipeline (fetch, decode, execute, memory access, write back). The processor supports three separate caches: a method cache, a stack cache and a data cache (D$) for heap allocated data. The stack cache in the Patmos processor is a special ring buffer on-chip memory. Compiler generated special instructions manage the accesses to the stack cache. Patmos compiler supports disabling the stack cache allocation. When stack cache is disabled, all the accesses go through the data cache. Therefore, once we disable the original stack cache, only the data cache is available, we

can use Patmos as a standard RISC processor with a data cache. To extend the processor with a stack cache without compiler support, we have to make a few changes in the pipeline. As we mentioned, to identify the stack accesses, we chose the method that uses the stack pointer register. Therefore, in the decode stage of the pipeline; a multiplexer determines whether the load/store is a stack load/store instruction. Each load/store instruction has two register operands, one source and one destination register. By comparing the effective address of the instruction with the stack pointer register, the type of the access is defined according to the output of the multiplexer and is passed through the execution stage to the memory stage. In the memory stage a multiplexer selects the appropriate cache (i.e., data or stack cache) to load/store the data:

Data Transfer with the Pipeline Patmos processor uses the OCPcache [84] to communicate with the caches. Therefore, we extended the multiplexer in the memory stage to transfer the OCPcache commands between the stack cache and the pipeline whenever the signal from the decode stage indicates that the accesses data is stack access.

Data Transfer with the Main Memory Since the Patmos processor's data cache sends and receives data to/from the main memory as well, it is important to merge the signals transferred between either the stack cache or data cache and the main memory (i.e., which of these components should send/receive command or data to/from the main memory at any given moment?). Therefore, we extended the merge component in the Patmos processor. The merge component is a simple multiplexer, passing the signals of the input with a valid (i.e. non-idle) request (read or write) and the data from the corresponding cache (on a write). On a read from the main memory, the merge component sends the requested data to the pipeline and the data is passed to the corresponding cache, i.e. the stack or data cache.

5.3 Summary

In this chapter we presented a mechanism to dynamically allocate stack frames to on-chip scratchpad memories (SPM). The proposed software stack cache is updated on function entry and exit to ensure that the stack frame of the active function is in the SPM. Using the SPM for stack data guarantees single cycle execution of loads and stores – always cache hits. This property can simplify worst-case execution time (WCET) analysis and also reduce the WCET bound. Moreover, we have proposed a stack cache in hardware for RISC style processors.

We discussed different aspects of the hardware managed stack cache, described the design and implementation of the proposed cache. Moreover, we have integrated our designs with the time-predictable processor Patmos. In chapter 6 We present the evaluation of the software managed and hardware managed stack cache in details.

Chapter 6

Evaluation

In this chapter we present the evaluation of our proposed ideas in previous chapters. We evaluate our designs in the context of the Patmos processor [87]. We provide the average case performance using Patmos' cycle accurate software simulator [24]. Moreover, whenever applicable, we report on the hardware resource consumption of our methods.

This chapter is structured as follows: in Section 6.1 we present the evaluation of the original stack cache design presented in chapter 3. Section 6.2 presents the evaluation of the lazy spilling stack cache design presented in chapter 4. We report on utilization of the block-aligned stack cache in section 6.3. We evaluate the virtual stack caching method in Section 6.4 and report on the number of TDM slots. In section 6.5 we present the measurements from the simulation of the software managed stack cache. Section 6.6 reports on the experiments evaluating the hardware managed stack cache. We finish the chapter by highlighting the main results from the evaluation of each proposed idea in the summary of Section 6.7. This chapter is based on the following published articles:

- *A Time-Predictable Stack Cache* [3].
- *Lazy Spilling for a Time-Predictable Stack Cache: Implementation and Analysis* [73].

- *Alignment of Memory Transfers of a Time-Predictable Stack Cache* [72].
- *Efficient Context Switching for the Stack Cache: Implementation and Analysis* [4].

6.1 Stack cache

We have implemented the stack cache within the Patmos processor in hardware and in the cycle accurate software simulator. In this section we report on the hardware size of our implementation. We present the evaluation of our implementation on FPGA platform. Moreover, we compare the stack cache with a standard cache. With the software simulation of Patmos we collect runtime statistics on stack and data cache usage with embedded benchmarks and provide the stack cache performance.

6.1.1 Hardware resource consumption

The Patmos processor is implemented on and FPGA. As we explained in chapter 3, we extended the Patmos processor with the stack cache. We synthesized the processor with the stack cache using the on Altera Cyclone II FPGA on the DE2-150 FPGA board. The stack cache consumes about 889 logic cells and the Patmos processor consumes about 15300 logic cells. Therefore, the overhead of the stack cache is very small. Adding the stack cache does not affect the processor frequency of 80 MHz.

The size of the stack cache is configurable and the default configuration is 64 32-bit words. Therefore, it consumes a single on-chip memory block.

Since the stack data is usually directed to the data cache, we look at the main differences between a normal data cache and the stack cache:

- A normal data cache needs hit detection by comparing the tag memory of each way with part of the address. This hit detection is usually on the critical path. It can be performed in parallel with data read, but needs to be performed before a write, which might add an extra cycle for a store instruction. With the stack cache we have guaranteed hits on load and store instructions and no need to compare with a tag memory. The equivalence to hit detection in the stack cache happens on `sres` and `sens`, which happen less often than loads and stores.

	Cache Size (KB)		
Line Size (Words)	1	2	4
2	1.4	1.2	1.1
4	2.7	2.3	2.2
8	5.3	4.7	4.3

Table 6.1: Total cache size in KB for different line sizes and cache sizes.

- A normal data cache needs a tag memory, which can consume a considerable amount of memory. In the stack cache only two pointers into the address space mark which data is in the cache and which data is only in the main memory.

From the latter we can see that the stack cache eliminates the need for the tag memory. Thus, we present a simple comparison between the data cache and the stack cache in terms of number of memory bits. In a standard cache, each cache line contains a tag word and a valid bit. The size of the tag word depends on the cache size, the cache line length. In set associative caches where the cache is divided into different sets, number of sets affects the size of the tag memory as well.

Table 6.1 shows the total memory size (tag and data memory) for different configurations of a direct mapped cache. We can see that the tag memory adds up to 40% to the memory consumption of the data cache. In contrast, the stack cache needs only the two registers to mark the address range that is in the cache. The two pointers also serve for the valid bit. Therefore, the total size of the stack cache is equal to size of the data memory used for a cache.

6.1.2 Cache performance

The stack cache design targets time-predictability and simple WCET analysis by splitting different data areas to specialized data caches. However, the split cache design can improve the average performance as well. We have collected the stack cache's average performance using the cycle accurate software simulator of Patmos [24]. The cycle accurate simulator allows us to collect the hit/miss rates for different stack cache configurations. Moreover, we can compare the performance of the stack cache with a standard data cache.

To evaluate the stack cache using the cycle accurate simulator, we compiled

Benchmark	Domain	LOC	Description
basicmath	automotive	665	Floating-point arithmetic
bitcnts	automotive	938	Count the number of set bits
qsort	automotive	47	Sort an input file of strings
csusan	automotive	2138	Corner finder - corner mode
esusan	automotive	2138	Corner finder - edge mode
ssusan	automotive	2138	Corner finder - smoothing mode
cjpeg	consumer	33735	Image compression
djpeg	consumer	33735	Image decompression
dijkstra	network	350	Shortest paths search
patricia	network	621	Routing using Patricia trees
dbf	security	2524	Data decryption using blowfish
ebf	security	2524	Data encryption using blowfish
drijndael	security	2442	Data decryption using AES
erijndael	security	2442	Data encryption using AES
sha	security	269	Secure hashing
crc-32	telecomm	291	Checksum calculation
fft	telecom	486	Fast-Fourier Transform
ifft	telecom	486	Fast-Fourier Transform
rawcaudio	telecomm	741	ADPCM audio coding
rawdaudio	telecomm	741	ADPCM audio decoding
rsynth	office	7042	
search-large	office	3054	Pratt-Boyer-Moore string search
search-small	office	460	Pratt-Boyer-Moore string search

Table 6.2: Short summary of benchmark programs used for the evaluation.

and executed a subset of the publicly available benchmark suite MiBench [35] on the Patmos simulator. These benchmarks cover a representative set of tasks often encountered in embedded systems, e.g. telecommunication and automotive domains. Table 6.2 summarizes the benchmark's characteristics.

The benchmarks were compiled to LLVM bitcode by the `clang` C frontend using optimization level `-O3` and then linked and optimized using `llvm-ld`. LLVM's `llc` tool generated Patmos machine code using two configurations: (1) with stack cache support enabled and (2) with stack cache support disabled, i.e., all the stack data is kept in the main memory. With the stack cache disabled, the normal data cache, caches the stack allocated data. The `gold` linker finalizes the code layout using a default memory layout, where all the code and data sections are placed into Patmos' main memory.

The Patmos simulator executes the benchmarks and is configured with a 1/4 KB stack cache, a 8 KB data cache, and a 64 KB instruction cache, organized

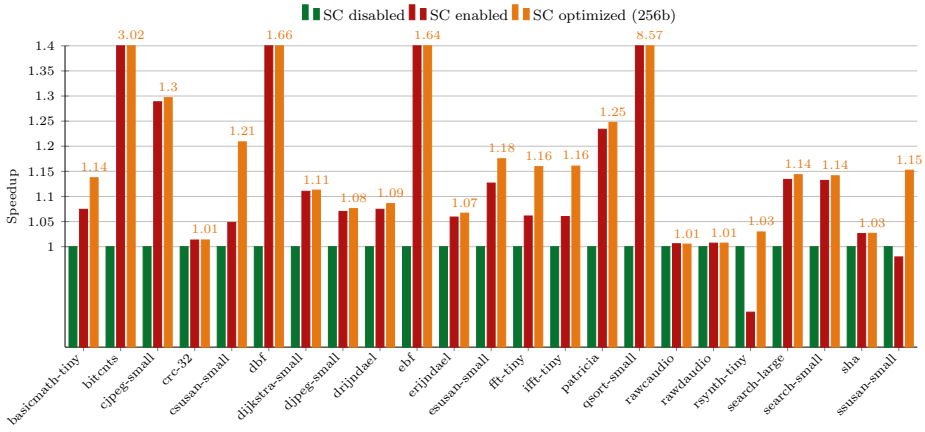


Figure 6.1: Speedup when running benchmarks with stack cache support disabled, enabled, and enabled with optimization (cache size 256 bytes).

as method cache [80]. A larger stack cache of 1 KB is big enough to cache stack data without the need to spill stack frames to the main memory. Therefore, we reduced the size of the stack cache to observe some spill and fill operations.

The stack cache is organized in word-sized blocks (4B), while the data and method caches are organized in 32-byte blocks. The 4-way set-associative data cache uses a least-recently-used (LRU) replacement policy and a write-through strategy with no-write allocation. The method cache likewise uses an LRU replacement policy. Transferring a cache block (32B) to or from main memory is assumed to take 40 cycles. Spills and fills are performed at word granularity.

MiBench offers a small and a large data set for most benchmarks. We run most of the benchmarks with the default data set. For some benchmarks we use smaller data sets. This is indicated by the suffix in the figures.

6.1.3 Run-time

In our first experiment we compare the total number of execution cycles for each benchmark with stack cache support enabled against a variant with the stack cache disabled. Enabling the stack cache allows us to (partially) allocate the stack frame of functions to the stack cache instead of the main memory. This reduces the number of accesses through the data cache, but at the same time increases the number of instructions, because dedicated instructions, emitted by

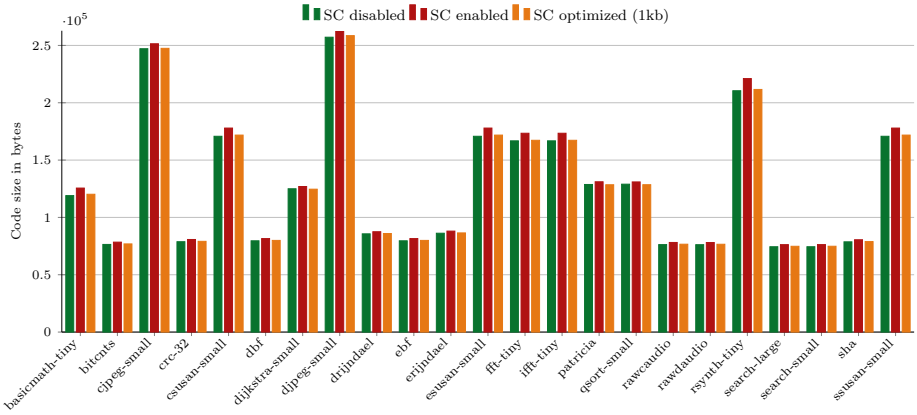


Figure 6.2: Code size in bytes for all benchmarks with stack cache support disabled, enabled and enabled with optimization (cache size 1 Kb).

the compiler, manage the stack cache explicitly.

As shown by Figure 6.1, enabling the stack cache reduces the total number of execution cycles for certain benchmarks. The gains are explained by (a) the additional cache space in the stack cache and (b) the handling of writes by the data cache. The additional cache space (1/4 KB) in the stack cache reduces the number of loads from the data cache and thus the number of accesses to the slow main memory. Additional gains are due to the long latency of stores to the data cache, which uses a write through strategy with no-write allocation.

Some benchmarks profit less from the stack cache, most notably the `crc-32`, `rawcaudio`, and `rawdaudio`. Loops that do not contain any spill code or function calls dominate these benchmarks. The simple stack cache allocation strategy thus cannot find any data to allocate to the stack cache.

We also evaluated the benefits of a simple compiler optimization that removes useless ensure operations (see SC optimized in Figure 6.1). An ensure instruction can be eliminated after a call whose worst-case stack cache occupancy combined with the ensure's size does not exceed the stack cache size. The runtime gain clearly out-weights the runtime overhead of managing the stack cache.

Figure 6.2 shows that the code size is only marginally increased by 1% in average and 1.3% in the worst-case. These numbers are expected to improve with more advanced stack cache allocation and code and data placement algorithms. Note that all benchmarks were compiled with full floating-point support enabled in the C library, which increases the overall code size.

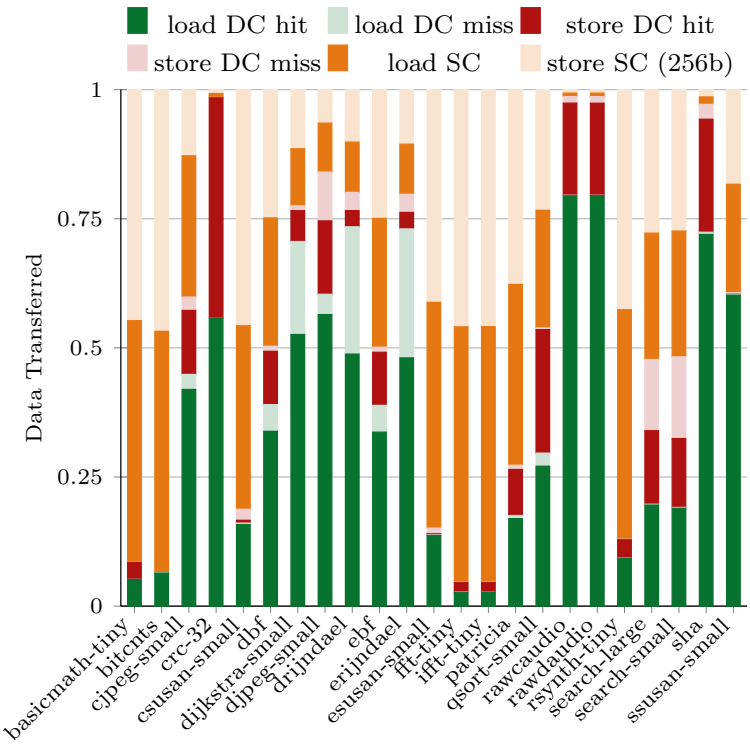


Figure 6.3: Normalized transfer volume of data accesses to the data and stack cache for each benchmark (data cache 8 KB, stack cache 256 bytes).

As we have seen, using the stack cache can be quite profitable. We thus present some additional data characterizing how the various benchmarks use the stack cache. Figure 6.3 shows the transfer volume (bytes read or written using load and store instructions) to and from the data and stack cache. The benchmarks that showed the best runtime improvements make heavy use of the stack cache. For instance, 79% of the memory accesses of the `bitcnts` go to the stack cache. With regard to data transfer volume, these numbers even increase. While roughly 65% of the accesses of the `patricia` benchmark target the stack cache, almost 75% of the transfer volume is serviced by the stack cache.

Benchmarks with little or no gain rarely make use of the stack cache, as shown by Figure 6.1. The main reason is that these benchmarks are dominated by a single loop without any spill code or function calls. This is confirmed by the numbers in Figure 6.4, which shows the amount of dynamically allocated

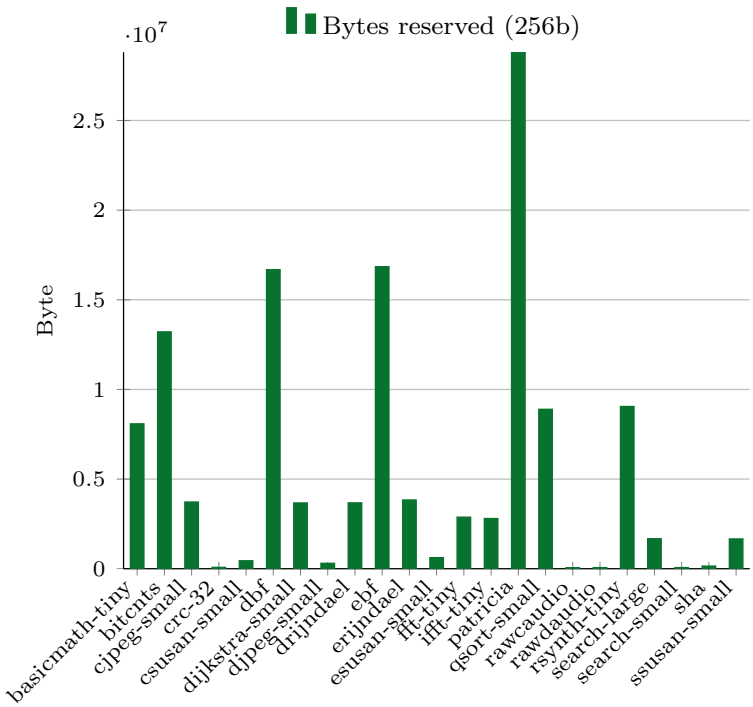


Figure 6.4: Data dynamically allocated on the stack cache. The stack cache usage correlates with the number of executed function calls (cache size 256 bytes).

data on the stack cache. As we can see, there is clear correlation between the amount of the data dynamically allocated on the stack and the execution time improvement. Moreover, we can see the correlation between the number of function calls (Figure 6.4) and the number of function calls executed in the benchmark (Figure 6.5).

The algorithm to allocate data on the stack cache only leverages compiler-generated spill slots, which are often linked to function calls (saving/restoring registers before and after calls). However, more powerful allocation algorithms will be able to overcome this limitation. For instance, the `rawcaudio` and `rawdaudio` benchmarks mostly operate on small buffers, which are potential candidates for stack cache allocation.

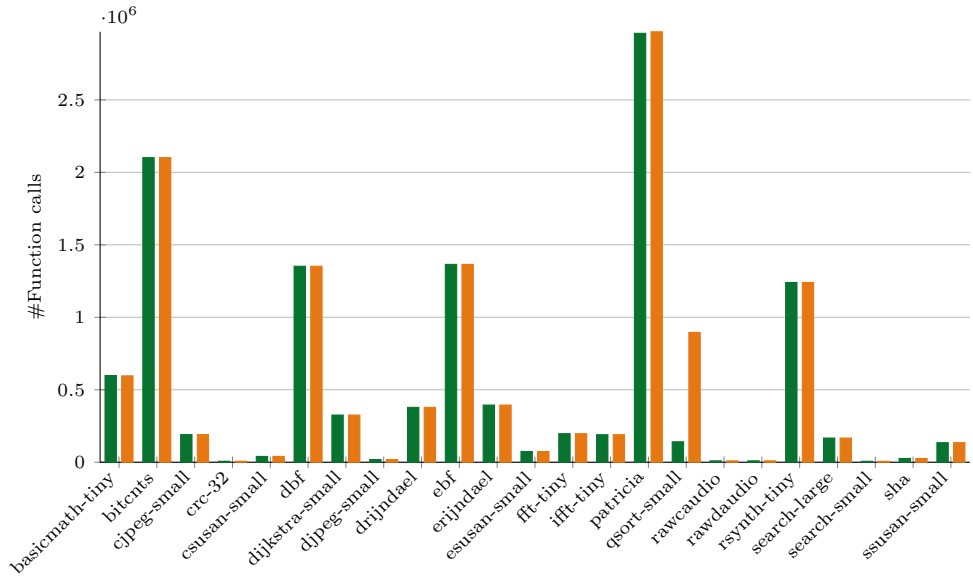


Figure 6.5: Number of dynamically executed function calls.

In the next section we provide the results of our experiments to evaluate the lazy spilling stack cache.

6.2 Lazy spilling stack cache

We use the Patmos processor for evaluating the impact of lazy spilling on hardware costs and average program performance as well. The hardware model of the processor was extended and statistics were collected on the speed and resource requirements after synthesis (Altera Quartus II 13.1, for Altera DE2-115). The average performance measurements were performed using the MiBench benchmarks suite. The programs were compiled using the Patmos LLVM compiler (version 3.4) with full optimizations (`-O3`) and executed on a cycle-accurate simulator. We compare five configurations utilizing (a) a standard data cache combined with a lazily spilling stack cache having a size of 128 or 256 bytes (LP_{128}, LP_{256}), (b) a standard data cache combined with a standard stack cache (SC_{128}, SC_{256}), and (c) a standard data cache alone (DC). The stack caches perform spilling and filling using 4 byte blocks. The data cache is configured to have a size of 2 KB, a 4-way set-associative organization, with 32 byte cache lines, LRU replacement, and a write-through strategy (recommended for

real-time systems [100]). In addition to data caches, the simulator is configured to use a 16 KB method cache for code. The main memory is accessed in 16 byte bursts and 7 cycles latency.

6.2.1 Implementation overhead

As we explained in Chapter 4, for lazy spilling, a single additional register is needed (`LP`). Updating the `LP` in the `sres` instruction adds two multiplexers to the original implementation. The `sfree` and stack store instructions each need an additional multiplexer. The area overhead is, therefore, very low. Moreover, these changes do not affect the processor's frequency (80 MHz).

6.2.2 Average performance

Table 6.6 shows the reduction in the number of blocks spilled in comparison to the standard stack cache. Note that results for `rawcaudio` and `rawdaudio` are shown as zero, as they never spill due to their shallow call nesting depth. The best result for LP_{128} is achieved for `bitcnts`, where lazy spilling practically avoids spilling. In the mean, spilling is reduced to just 17%. The worst result is attained for `qsort-small`, where 62% of the blocks are spilled. For LP_{256} spilling is reduced to 30% in the mean. The best result is observed for `search-large`, where essentially all the spilling is avoided. For `crc-32`, `drijndael`, `erijndael`, and `sha` only marginal improvements are possible, since these benchmarks already spill little in comparison to the other benchmarks. The worst result of those benchmarks with a relevant amount of spilling is obtained for `qsort-small`, where 76% of the blocks are spilled.

Since miss rates are not suitable for comparison against standard data caches, we compare the number of bytes accessed by the processor through a cache in relation to the number of stall cycles it caused, i.e., $\frac{\#RD+\#WR}{\#Stalls}$. A high value in Table 6.7 and Table 6.8 means that the cache is efficient, as data is frequently accessed without stalling. The data cache alone gives values up to 3.3 only. Ignoring benchmarks with little spilling, the best result for SC_{128} is achieved by `dbf` (477.4). For SC_{256}, `bitcnts` gives the best result (17054.7). Lazy spilling leads to consistent improvements over all benchmarks. An interesting observation is that for most benchmarks the presence of a stack cache *improves* the performance of the data cache. The best example for this is `bitcnts`, but also `csusan` and `ssusan` profit considerably, where the data cache alone delivers 1.2 bytes per stall cycle. When a stack cache is added to the system, this value jumps up to 192.4 and 196.1 respectively.

Benchmark	LP_{128}	LP_{256}
basicmath-tiny	0.17	0.53
bitcnts	0.00	0.71
cjpeg-small	0.51	0.09
crc-32	0.03	1.00
csusan-small	0.16	0.72
dbf	0.47	0.00
dijkstra-small	0.20	0.54
djpeg-small	0.34	0.66
drijndael	0.20	1.00
ebf	0.44	0.00
erijndael	0.57	1.00
esusan-small	0.25	0.02
fft-tiny	0.08	0.56
ifft-tiny	0.08	0.56
patricia	0.27	0.55
qsort-small	0.62	0.76
rawcaudio	0.00	0.00
rawdaudio	0.00	0.00
rsynth-tiny	0.08	0.48
search-large	0.48	0.00
search-small	0.49	0.02
sha	0.20	0.91
ssusan-small	0.20	0.80

Figure 6.6: Reduction in spilling for the lazy spilling stack cache with different sizes.

Figure 6.9 shows the total normalized execution cycles for two different lazy spilling stack cache sizes, i.e. 128 and 256 bytes. As we can see, increasing the lazy spilling stack cache's size shows little improvement compared to a 128 bytes cache. Thus, we can gain performance improvements using only a small 128 bytes cache.

Our measurements show that lazy spilling eliminates most spilling in comparison to a standard stack cache. Moreover, the efficiency of the standard data cache is improved in many cases. Due to the low memory latency assumed here, this translates to run-time gains of up to 21.8% in comparison to a system with a standard stack cache. In the mean, the speedup amounts to 8% and 9.2% in comparison to a system only equipped with a data cache.

In the next section we provide the results of our experiments to evaluate the block-aligned stack cache.

Benchmark	LP_{128}	SC_{128}	LP_{256}	SC_{256}
basicmath-tiny	2.3	4.0	26.4	34.0
bitcnts	4.6	12.2	17054.7	19201.4
cjpeg-small	116.9	148.4	3470.7	6154.4
crc-32	9.0	21.3	814.9	814.9
csusan-small	11.3	18.6	1218.8	1430.0
dbf	477.4	623.0	–	–
dijkstra-small	19.5	32.8	335.2	433.7
djpeg-small	9.0	13.5	293.4	361.5
drijndael	15.8	28.7	185620.0	185620.0
ebf	172.5	224.6	–	–
erijndael	32.6	43.3	258340.0	258340.0
esusan-small	15.9	25.3	70.7	139.5
fft-tiny	3.1	5.8	85.0	103.4
ifft-tiny	3.1	5.9	83.1	101.0
patricia	2.5	4.2	26.4	31.9
qsort-small	3.1	3.7	7.8	8.6
rawcaudio	–	–	–	–
rawdaudio	–	–	–	–
rsynth-tiny	16.0	29.9	1096.1	1539.8
search-large	2.9	3.9	26.3	52.5
search-small	2.9	3.7	28.1	54.8
sha	8.3	14.1	668.7	700.6
ssusan-small	29.2	43.9	4313.5	4678.0

Figure 6.7: Bytes accessed per stall cycle for the stack cache and lazy spilling stack cache.

6.3 Block aligned stack cache

For our experiments we extended the hardware implementation of the stack cache available with the Patmos processor as well as the cycle-accurate simulation infrastructure and the accompanying LLVM compiler (version 3.4). The average case performance was measured for all benchmarks of the MiBench benchmark suite. The benchmaks were compiled, with optimizations (`-O2`) and stack cache support enabled, and then executed on the Patmos simulator to collect runtime statistics. The simulator was configured to simulate a 2 KB data cache (32 B blocks, 4 way set-associative, LRU replacement policy, and write-through strategy), a 2 KB method cache (32 B blocks, associativity 8), and a 128 B stack cache (with varying configurations). All caches are connected to a shared main memory, which transfers data in 32 B bursts. A moderate access latency of 14 cycles for reads and 12 cycles for writes is assumed.

The benchmarks were tested under three different scenarios: (1) the stack cache performs unaligned memory transfers (`unaligned`), (2) the compiler generates

Benchmark	LP_{128}	SC_{128}	LP_{256}	SC_{256}	DC
basicmath-tiny	1.1	1.1	1.1	1.1	1.1
bitcnts	191.6	191.6	193.7	193.7	1.2
cjpeg-small	1.0	1.0	1.0	1.0	1.1
crc-32	0.9	0.9	0.9	0.9	0.9
csusan-small	2.2	2.2	2.3	2.3	1.5
dbf	1.0	1.0	1.0	1.0	1.0
dijkstra-small	1.4	1.4	1.4	1.4	1.4
djpeg-small	0.8	0.8	0.8	0.8	0.8
drijndael	0.9	0.9	0.9	0.9	0.9
ebf	1.0	1.0	1.0	1.0	1.0
erijndael	0.9	0.9	0.9	0.9	0.9
esusan-small	3.4	3.4	3.6	3.6	1.5
fft-tiny	1.1	1.1	1.1	1.1	1.1
ifft-tiny	1.2	1.2	1.1	1.1	1.1
patricia	1.0	1.0	1.0	1.0	1.0
qsort-small	1.0	1.0	1.0	1.0	1.0
rawcaudio	0.7	0.7	0.7	0.7	0.7
rawdaudio	1.1	1.1	1.1	1.1	1.1
rsynth-tiny	1.9	1.9	1.9	1.9	1.3
search-large	0.8	0.8	0.8	0.8	0.9
search-small	0.8	0.8	0.8	0.8	0.9
sha	1.6	1.6	1.6	1.6	1.6
ssusan-small	17.1	17.1	17.1	17.1	3.3

Figure 6.8: Bytes accessed per stall cycle for the data cache with stack cache and lazy spilling stack cache enabled and disabled.

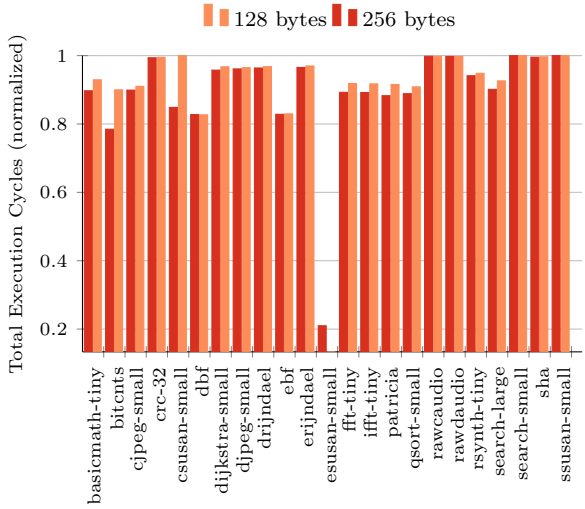


Figure 6.9: Normalized execution cycles for two different configurations of the lazy spilling stack cache, 128 and 256 bytes.

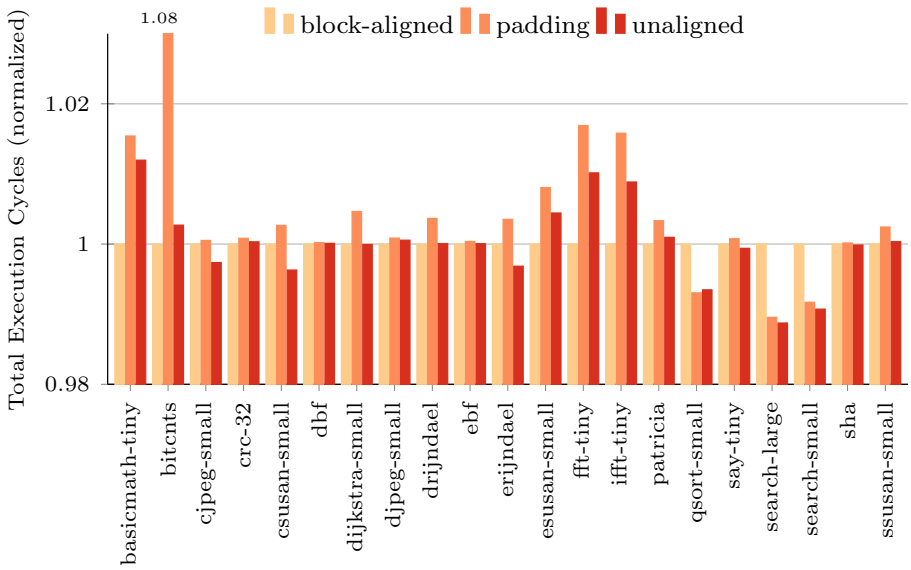


Figure 6.10: Total execution cycles normalized to the `block-aligned` configuration (lower is better).

suitable *padding* to align all stack allocations and consequently all memory transfers (`padding`), and (3) the stack cache employs the block-aligned strategy (`block-aligned`).

Stack data is usually aligned at word boundaries for Patmos, which applies to the `unaligned` and `block-aligned` configurations. The `padding` configuration, however, aligns all data with the burst size (32 B).

The runtime impact of the various strategies to handle the alignment of memory transfers between the stack cache and the main memory is summarized in Figure 6.10. Overall, the `unaligned` and `block-aligned` configurations are very close with respect to runtime, while the `padding` configuration performs the least. In particular, the `bitcnts`, `basicmath-tiny`, `fft-tiny`, and `ifft-tiny` benchmarks here show runtime increases of 2% and more.

Note that the runtime contribution of the stack cache is relatively small, which in general precludes very large variations in the total runtime due to the stack cache. The simulator thus was extended to collect detailed statistics on the number of stall cycles induced by the stack cache as well as the spilling and filling performed. Figure 6.11 shows the total number of stall cycles induced by the stack cache, normalized to the `block-aligned` configuration. The `padding` configuration increases the number of stall cycles in all cases in relation to

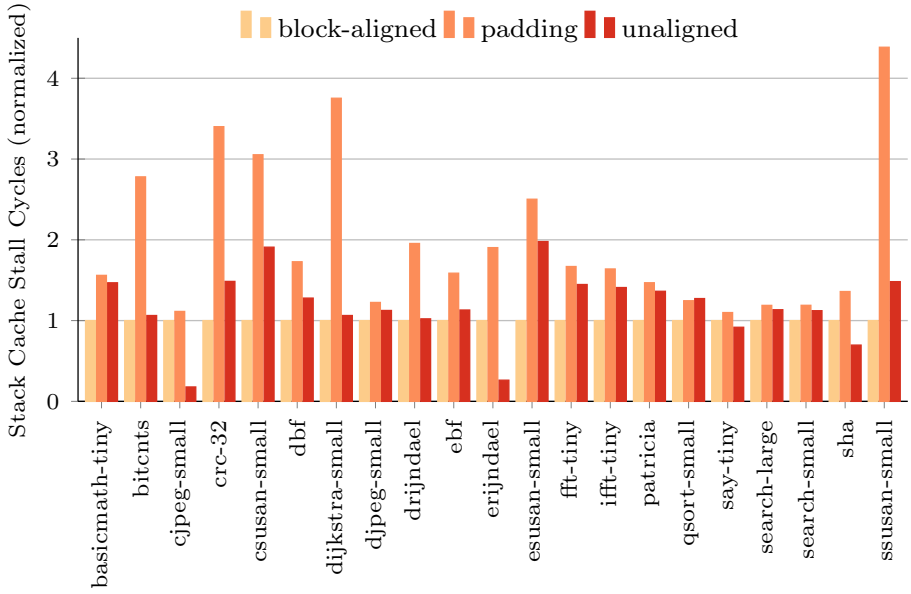


Figure 6.11: Total number of stall cycles induced by the stack cache normalized to the `block-aligned` configuration (lower is better).

our `block-aligned` strategy (up to a factor of more than 4). The padding introduced by the compiler generally increases the stack cache's occupancy and consequently leads to additional memory transfers. Also, for the `unaligned` configuration the number of stall cycles is larger than our new strategy, since the small unaligned memory transfers performed by this configuration induce some overhead. For two benchmarks, `cjpeg-small` and `erijndael`, the number of stall cycles is considerably smaller in this configuration. Our `block-aligned` stack cache here suffers additional filling and spilling due to its reduced effective size, as shown in the following Table.

The impact of the various configurations on the amount of data spilled and filled from/to the stack cache is shown in Table 6.12. As noted above the `padding` configuration performs additional memory transfers (spills and fills) due to the padding introduced by the compiler to ensure alignment. The `unaligned` configuration on the other hand requires the least filling and spilling as it transfers the precise amount of data needed. In addition, the reduced stack cache size available for the `block-aligned` strategy (recall that one block is reserved as an alignment buffer) plays in favor of the `unaligned` configuration.

To summarize, compiler generated padding is a simple solution to the alignment problem for the stack cache, which is easy to analyze and generally performs

	Padding		Unaligned	
Benchmark	Spill rel.	Fill rel.	Spill rel.	Fill rel.
basicmath-tiny	2.14	2.05	1.30	1.20
bitcnts	3.00	3.00	0.69	0.69
cjpeg-small	1.17	1.27	0.13	0.20
crc-32	3.77	11.68	0.99	0.94
csusan-small	3.50	3.59	1.46	1.36
dbf	1.87	1.60	0.87	0.85
dijkstra-small	4.82	4.80	0.50	0.51
djpeg-small	1.20	1.25	0.52	0.53
drijndael	1.96	1.76	0.18	0.23
ebf	1.73	2.35	0.82	1.11
erijndael	1.87	1.71	0.17	0.23
esusan-small	3.05	3.11	1.61	1.55
fft-tiny	2.08	2.16	1.29	1.28
ifft-tiny	2.06	2.15	1.28	1.27
patricia	1.71	1.76	0.96	0.98
qsort-small	1.75	1.88	1.20	1.16
say-tiny	1.16	1.28	0.38	0.33
search-large	1.49	1.66	0.96	1.14
search-small	1.49	1.69	0.93	1.09
sha	1.59	1.11	0.36	0.07
ssusan-small	3.65	3.74	0.89	0.86

Figure 6.12: Normalized words spilled and filled by the stack cache configurations `padding` and `unaligned` in comparison to the `block-aligned` configuration).

reasonably well, but may suffer from bad outliers. It generally leads to increased spilling and filling as well as a reduced utilization of the stack cache. Generating unaligned memory transfers naturally performs well for the average case, but, complicates WCET analysis since the alignment of the stack data is highly context dependent. The new solution proposed in this work, the block-aligned stack cache, offers a reasonable trade-off, which combines moderate hardware overhead with good average-case performance and simple WCET analysis.

In the next section, we present the evaluation of the virtual stack caching method.

6.4 Virtual stack caching

This section presents an evaluation of the preemption costs associated with the stack cache, covering both results from the measurements related to the proposed hardware extension to virtualize the stack cache.

The benchmarks are taken from the MiBench benchmark suite [35], which covers a large variety of small- and medium-sized programs typically found in embedded systems. The programs were compiled with optimizations enabled (`-O2`) using the LLVM[1] compiler for the Patmos processor [87]. The hardware of the platform is configured with a 64KB, 4-way set-associative data cache using LRU replacement, and a write-through policy (recommended for real-time systems [100]). Code is cached by a 64KB method cache [87] with LRU replacement and 32 code block entries. The stack cache is 256b small and uses a lazy pointer [73]. Note that varying the stack cache size between 256b and 1KB showed little impact on the results obtained. The global memory is assumed to have a moderate latency of 21 cycles. Memory transfers are performed in bursts of 32b. The cache line size of all caches matches the memory's burst size. Note, the stack control instructions still operate in words, i.e., stack frame sizes have to be word-aligned, while memory transfers are performed in bursts.

6.4.1 Unused TDM slots

Using the cycle-accurate Patmos simulator we collected statistics on the number of unused TDM (or bus) slots during the execution of a benchmark in order to perform context saving/restoration of preempted tasks along with the execution of another task. We compare three multi-core configurations with 2, 9 and 16 cores shown in Table 6.3.

The measurements are encouraging, in particular when the number of cores is small. For a given fixed interval of time, when the number of cores increases, the number of pre-assigned TDM slots per core decreases. This explains the drop in the percentage of free TDM slots for the 9 and 16 cores configuration. However, if we consider these numbers, a large amount of slots are available, as the total execution time increases. Most of the benchmarks would therefore easily allow transferring several virtual stack caches to and from main memory during their execution. Benchmarks with a low share of free slots, in particular `erijndael` and `drijndael`, can be considered outliers. These benchmarks suffer from particularly bad cache miss rates of more than 35% even with the large 64KB data cache. We do not expect such bad miss rates in a realistic system and thus conclude from these average-case measurements that it is feasible to hide context switch costs by transparently using TDM slots left unused by the currently running task. Further work is needed to judge whether these measurements actually can be translated to bandwidth guarantees and used within the schedulability analysis, as discussed in Section 4.3.3.

[1] `http://www.llvm.org/`

Benchmark	2 cores	9 cores	16 cores
basicmath-tiny	24.58	4.33	1.86
rawcaudio	59.27	1.30	0.71
dijkstra-small	18.42	0.29	0.13
ebf	15.70	0.07	0.01
qsort-small	12.44	0.35	0.11
rawdaudio	58.20	8.10	4.74
search-large	10.55	0.81	0.02
djpeg-small	14.14	0.89	0.01
csusan-small	54.12	15.42	8.73
crc-32	19.75	0.00	0.00
ssusan-small	86.96	58.74	44.23
ifft-tiny	41.55	10.84	5.30
erijndael	14.36	0.14	0.00
cjpeg-small	32.49	4.18	2.10
search-small	9.59	0.72	0.00
sha	34.87	4.05	2.05
drijndael	16.63	0.18	0.00
say-tiny	66.47	25.06	11.92
bitcnts	99.63	98.21	96.58
fft-tiny	41.89	10.89	5.41
dbf	15.67	0.07	0.00
patricia	26.68	4.58	2.09
esusan-small	67.24	27.04	15.93

Table 6.3: Percentage of free slots w.r.t. the total number of TDM slots required to execute the benchmarks on multi-cores.

6.4.2 Hardware evaluation

The impact of the hardware modifications[2] on the hardware was evaluated using Altera Quartus II 13.1 tool suite targeting an Altera DE2-115 board. The results show an area overhead of only 1.8% (which does not reflect the gains due to the merged memories). Moreover, the processor frequency drops from 82 MHz to 81 MHz, due to the longer combinatorial path for the address computation. Our design at this stage is not optimized and distributing the combinatorial logic for the address calculation to different stages of the pipeline can improve the frequency. We thus conclude that the virtual stack caches cause very little overhead.

In the following two sections we discuss the evaluation of the software managed stack cache and hardware managed stack cache respectively.

[2]The modulo operator is in practice a simple bit mask.

6.5 Software managed stack cache

To evaluate the effect of the software stack cache, we explore several scenarios with different stack cache sizes and a fixed data cache size of 2 KB.

Figure 6.13 shows loads and stores to different memory areas per 100 instructions executed. We present the numbers for three configurations: one configuration with no stack cache, one with a stack cache of size 256 bytes, and one with a stack cache of size 2 KB. The scratchpad loads and stores show the accesses to the software stack cache. Uncached loads and stores are accesses to the main memory due to spill and fill operations. Data cache loads and stores represent accesses to heap allocated data or the shadow stack.

Figure 6.13 shows most of the benchmarks can benefit even using a very small stack cache (256B). For instance, almost 80% of the accesses of the `basicmath` benchmark go to the stack cache. Furthermore, we see only a few uncached loads and stores in any of the benchmarks. In the case of the 2 KB stack cache, we can see total elimination of the data transfer to the main memory due to spilling and filling in most of the benchmarks. It should be noted that using the software stack cache increases the number of instructions in total. Therefore, the cumulative accesses to the stack and data caches are different in the three configurations.

6.6 Hardware managed stack cache

As we explained, to evaluate the efficiency of a stack cache for RISC style processors, we use the Patmos processor. We evaluate the average performance of the processor using the stack cache. Firstly, we extended the hardware model of the processor and synthesized it using Altera Quartus II 13.1, for Altera DE2-115 board. The synthesis results show that without the stack cache Patmos consumes almost 15000 logical cells. The stack cache increases the number of logical cells to almost 16000. Therefore, the area overhead of the stack cache is almost 6.3%. Moreover, our changes do not affect the processing frequency, i.e. 80 MHz. Secondly we measure the average performance of the processor using the MiBench benchmark suite. All programs are compiled with the Patmos LLVM compiler. We performed measurements using the cycle accurate simulator `pasim` of the Patmos processor. The Patmos processor provides performance counters for caches. However, to utilize the performance counters, we have to adapt the source codes of every benchmark. This is a non-trivial task, therefore, to facilitate our experiments, we chose the cycle accurate simulator

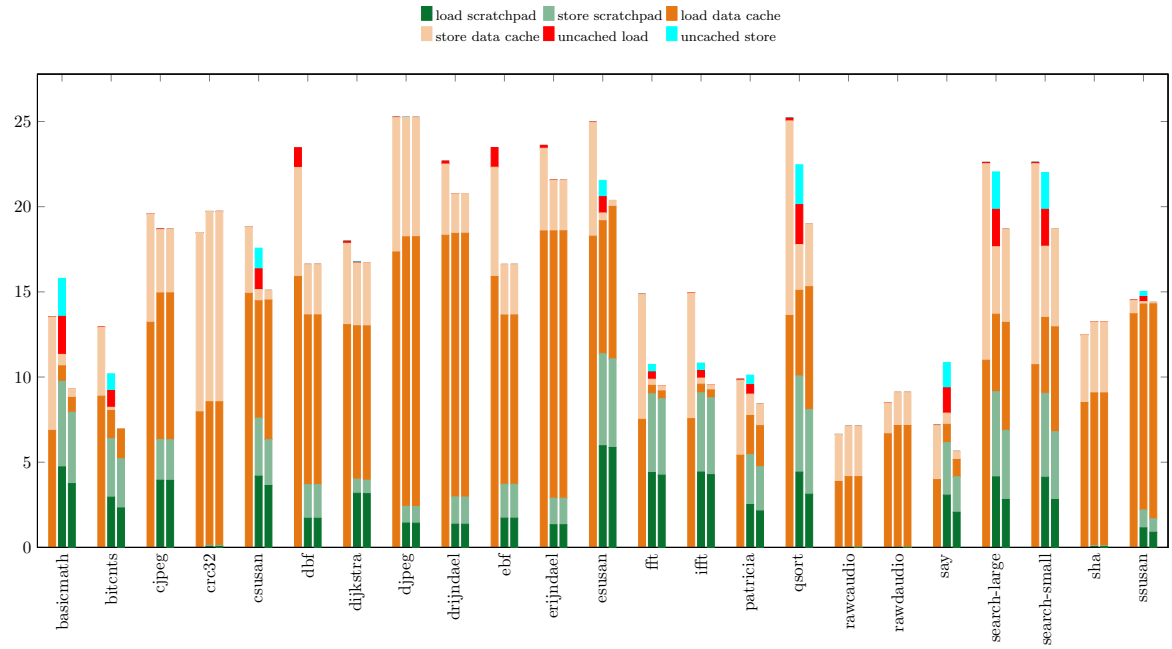


Figure 6.13: Memory accesses by memory type per 100 instructions (within each group: no stack cache, stack cache of size 256 B, and stack cache of size 2 KB).

for our average performance measurements.

MiBench contains small and large data set for most of the benchmarks. We run the benchmarks with the default data set, the small and tiny inputs. For the benchmarks that we use smaller data sets, we indicate this by the suffix in the figures.

Our simulations use a standard data cache. The data cache is 2-way set-associative cache with LRU replacement policy. The data cache size is set to 2 KB. We explore the effect of the stack cache type and size on average-case performance with changing simulator's configuration. In addition to data caches, the simulator uses a 16 KB method cache for code. The main memory has 7 cycles access latency with 16 byte bursts transfers, i.e. it takes 7 cycles to transfer 4 words between the main memory and the cache. The stack cache size is set to 256 bytes. For both the stack cache and the data cache we set the line size to 4 words.

We consider three different configurations for our experiments: The first configuration uses a direct mapped stack cache with write through writing policy and a write through data cache. The second configuration uses a 2-way set-associative stack cache with LRU replacement policy with write through writing policy and a write through data cache. The third configuration uses a direct mapped stack cache with write back writing policy and a write back data cache.

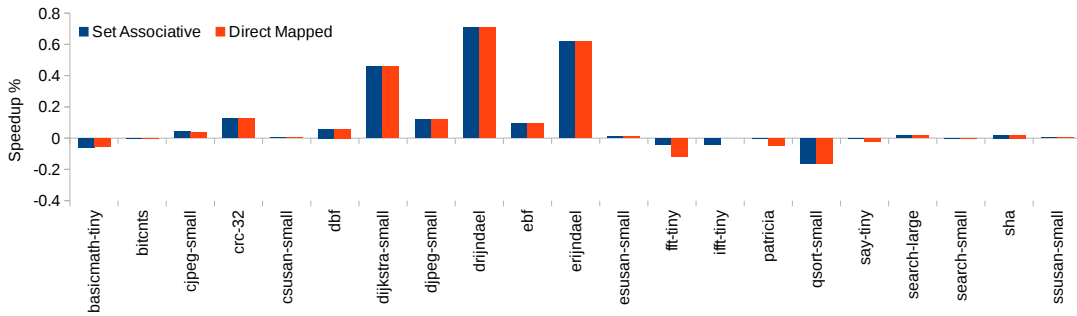


Figure 6.14: Speedup using the stack cache with two different configurations, 2 ways set associative and direct mapped stack cache.

6.6.1 Average performance

Figure 6.14 shows the speedup of each benchmark in presence of the stack cache. The figure presents two different configurations for the stack cache: one configuration with a 2 way set associative stack cache and one with a direct mapped stack cache. Both of the configurations use the write through writing strategy. Note that results for `rawcaudio` and `rawdaudio` are not shown since these benchmarks have shallow nesting call depth and therefore cannot utilize the stack cache.

As we can see, using the write through writing strategy, splitting the stack data accesses to a dedicated stack cache improves the performance for most of the benchmarks. Moreover, figure 6.14 shows the same speedup for both cache configurations for all the benchmarks and thus the simulation results are in line with our expectations that the stack cache design does not benefit from set associativity. This is an important observation with regard to the implementation cost. Since the direct mapped cache requires less hardware resources than the set associative cache. Therefore, direct mapped cache is the best way of implementation of the stack cache.

We should note that benchmarks with negative speedup cannot utilize the small stack cache due to large number of stack data accesses in these benchmarks. For example, in the `quicksort` benchmark, most of the accesses go to the stack cache. Therefore, a 256 bytes stack cache, has more misses than a 2K data cache and results in lower performance. However, increasing the stack cache size from 256 bytes to 1K shows 0.02% speedup for this benchmark. Increasing the size of the stack cache to 2K does not increase the performance anymore.

Figure 6.15 shows the speedup using a write-back stack cache and write back

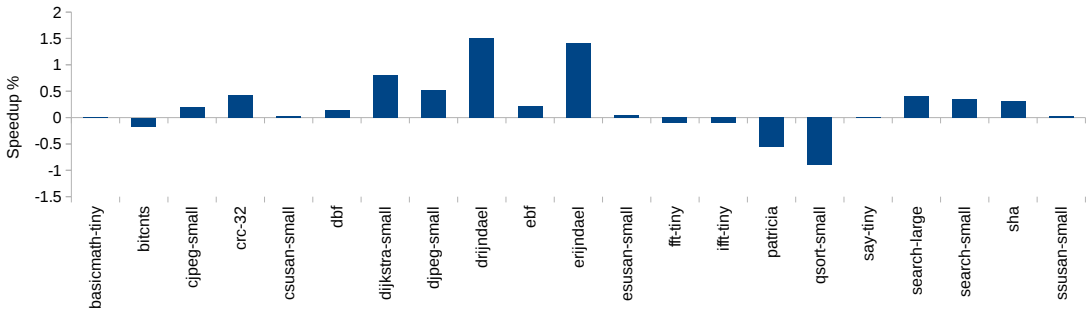


Figure 6.15: Speedup with write back stack cache.

data cache. As we can see, the write back cache improves the utilization of the stack cache considerably in most of the benchmarks. These results are in line with our assumption that the stack cache performs better using a write back writing strategy.

Our measurements show with a small stack cache, we can improve the average performance for most of the benchmarks. Moreover, to minimize FPGA logic and memory utilization, on-chip memories with sizes that are a power of 2 bytes are best design choices. Implementing memories with sizes that are not powers of 2 can result in inefficient memory and logic use. Therefore, to improve the performance, using only the data cache, we need to double the size of the data cache.

6.7 Summary

In this chapter we presented the evaluation of the idea of splitting the data cache for improving the time predictability, in hardware and using cycle accurate simulations.

We have implemented the stack cache in hardware in the time predictable processor Patmos. Furthermore, benchmarking of standard embedded benchmarks showed that even a small stack cache provides a good hit rate.

Moreover, we evaluated our proposed methods of increasing the performance of the time predictable stack cache. Our measurements show that the lazy spilling method eliminates most spilling in comparison to a standard stack cache, leading to run-time gains of up to 21.8% in comparison to a system with a standard stack cache. The block-aligned stack cache, offers a reasonable trade-off between the compiler generated padding and unaligned memory transfers. This method combines moderate hardware overhead with good average-case performance and simple WCET analysis.

Our measurements on virtual stack caching method indicated that with a small overhead we can easily transfer several stack caches for typical programs and reduce the context switching overhead.

The software managed stack cache shows that most of the benchmarks can benefit even using a very small stack cache (256 bytes). Furthermore, we see only a few uncached loads and stores in any of the benchmarks. Moreover, the evaluation of the hardware managed stack cache, shows for an area overhead of 6.3% with 80 MHz frequency we can gain up to 1.5% of run-time gain.

Chapter 7

Conclusion

In this thesis we have proposed design of a time predictable cache for caching stack allocated data in embedded systems, the *stack cache*. We integrated our design with the Patmos time predictable processor to investigate the performance improvements gained from the stack caching. Moreover, we extended our design with two different methods for improving the performance further: the *lazy spilling* stack cache and the block-aligned stack cache. Then, we proposed virtualizing stack caches to avoid context switching overhead for the stack cache design. Moreover, we extended our design to a broader range with proposing a time-predictable software managed stack cache and a hardware managed stack cache for RISC style processors. Our implementations are open source and can be downloaded from Patmos' repository. We have reported results of our experiments from implementing the hardware and cycle accurate simulations of proposed methods and ideas.

7.1 Main Results

The growing complexity of modern computer architectures, increasingly complicates the prediction of the run-time behavior of embedded systems. For real-time systems, where a safe estimation of the program's WCET is needed, time-predictable architectures are necessary. Caches as essential parts of any

embedded system. For systems with hard deadlines, we should be able to provide a tight WCET and thus, we need the memory and caches to be analyzable. Meaning, we need time-predictable cache designs. Cache analysis tries to predict hits and misses statically for a precise WCET estimation [26]. We consider three different types of data, global static data, stack allocated data, and heap allocated data. Global static data is easy to analyze since their addresses are determined during program linking. Heap allocated data is the most difficult type to analyze since the addresses of objects are unknown before running the program [77].

The stack data for a program is stored in the stack memory area and besides the return address information and callee saved registers, contains function local variables and data structures. As the access frequency on this data area is very high, the stack data benefits from caching. A WCET analysis tool can statically determine the addresses of stack allocated data when the call tree can be determined and when there is no dynamically sized allocation on the stack. We used these characteristics to design a new cache to improve the WCET of the data caches and the overall WCET of the system that uses data caches.

In this work, we described the idea and implementation of a time-predictable cache for stack allocated data: the *stack cache*. The stack cache is designed to simplify the WCET analysis and to improve the WCET. The stack cache is implemented as a kind of ring buffer with two pointers: *stack top* (`sc_top`) and *memory top* (`m_top`). The former points to the top of the logical stack and the latter points to the top element present in the main memory. Three stack control instructions (`sres`, `sens`, `sfree`) manipulate the two stack pointers and initiate the corresponding memory transfers as needed (spill and fill for `sres` and `sens` respectively). Using these three instructions, the compiler generates code to manage the stack frames of functions, quite similar to other architectures with exception of the ensure instruction. From the evaluation we have seen that even a very small stack cache (1 KB and less) serves very well for caching and provides good hit rates. With static data flow analysis it is possible to determine the value of the two pointers `sc_top` and `m_top` and thus, the exact spill and fill points and sizes in the stack cache for providing a precise WCET.

We looked at two different optimization methods to increase the performance of the stack cache by reducing the transfers between the stack cache and the main memory.

The first method, lazy spilling, is based on the fact that in many cases the data spilled to the main memory has the exact same value as the data already stored in the main memory. This may happen in situations where data is repeatedly spilled, e.g., due to calls in a loop, but not modified in the meantime. Thus, the main idea is to track of the data that is coherent between the main memory

and the stack cache. The cache can then avoid the needless spilling of the coherent data. We showed that this tracking can be realized efficiently using a single pointer, the so-called *lazy pointer*. Our measurements showed that lazy spilling eliminates most of the spilling in comparison to a standard stack cache. Moreover, an increase in the efficiency of the standard data cache is observed in most of the benchmarks.

The second method, the block-aligned stack cache is based on the fact that for WCET analysis, alignment of the stack cache content needs to be known or otherwise all access have to be assumed unaligned. Thus, we suggest using a burst-sized block of the stack cache as an *alignment* buffer. The alignment buffer makes sure that the addresses of all the transfers between the main memory and the stack cache are aligned to the main memory controller's burst size. Although this approach reduces the effective stack cache size by one block, it allows us to perform allocations at word granularity, thus, improving the cache's utilization. This method has a simple implementation of `sres` and `sens`, while `sfree` requires some minor extensions. Our measurements showed good average case performance for the block-algined stack cache compared to other methods of alignment.

In addition to the above, we discussed that the simple structure of the stack cache has drawbacks. When multiple tasks are executed using preemptive scheduling, the two pointers capture the cache state of the currently running task. Thus, the states of other (preempted) tasks are lost due to `sc_top` and `m_top` being overwritten. As a consequence, the entire stack cache content has to be *saved* to main memory when a task is preempted. In addition, the stack cache content has to be *restored* before that task is resumed. This may induce considerable overhead that has to be accounted for during the analysis of a real-time system equipped with a stack cache. To solve this problem we proposed a hardware extension to *virtualize* several stack caches. Virtual stack caching allows us to quickly switch between caches.

We implemented our design in Patmos processor with very low hardware overhead and showed that the preemption overhead can partially be hidden and thus, profits the Worst-Case Response Time (WCRT) of the preempting task.

Finally, we looked at the stack caching from two different angles: a software managed and a hardware managed stack cache. The software managed stack cache provides an alternative for our time-predictable stack cache for systems where changing the hardware is not an option. The software managed stack cache presents a mechanism to dynamically allocate stack frames in on-chip scratchpad memories (SPM). The software stack cache is updated on function entry and exit to ensure that the stack frame of the active function is in the SPM. Using the SPM for stack data guarantees single cycle execution of loads and

stores – always cache hits. This property can simplify worst-case execution time (WCET) analysis and also reduce the WCET bound. The hardware managed stack cache, on the other hand, relies only on hardware to identify accesses to stack data and direct them to a dedicated cache. Thus, this design requires no changes in the ISA and compiler. We explored different characteristics of the stack cache targeting the RISC style processors and showed that even with small cache sizes we gain performance improvements.

7.2 Future Directions

During this work we have examined many aspects of time-predictable stack caching for embedded systems. However, there are some ideas that can provide insights for future work. In this section we present some of these ideas or issues that are interesting to examine in future work.

7.2.1 Analysis of free TDM slots

In Chapter 4, we showed the number of unused TDM (or bus) slots during the execution of a benchmark in order to perform context saving/restoration of preempted tasks along with the execution of other tasks. One problem to solve is statically determining whether the number of free slots is enough for transferring all the virtual stack caches' content to the main memory. Another point to follow is that we can evaluate the behavior of the stack cache, in particular the dynamic partitioning of the virtual caches for the minimal WCET with regard to the tasks priorities and their respective WCET.

7.2.2 Extending the stack caching idea

In Chapter 2, we discussed about several methods to improve the performance of the data caches based on the type of the reference to the memory. For example, cache locking [94] uses a compiler technique to lock the data in the cache. It would be interesting to investigate the integration of this method with the stack caching. The stack caching proposed for the RISC style processors is a good start for applying these techniques. Since there are no worries about the time-predictability and these techniques are mostly targeting the improvement of the average performance. However, for any further improvement of the time-predictable stack cache, we have to keep time-predictability as the fundamental

characteristic in mind. One possible extension of the time-predictable stack cache can be [91]. However, this method requires compiler support.

Bibliography

[1] *TC1130 User's Manual*.

[2] T-crest. `http://www.t-crest.org/`, 2012-2014.

[3] S. Abbaspour, F. Brandner, and M. Schoeberl. A time-predictable stack cache. In *Proc. of the Workshop on Software Technologies for Embedded and Ubiquitous Systems*. 2013.

[4] Sahar Abbaspour, Florian Brandner, Amine Naji, and Jan Mathieu. Efficient context switching for the stack cache: Implementation and analysis. In *to appear in RTNS 2015: 23rd International Conference on Real-Time Networks and Systems*, Lille, France, November 4-6, 2015.

[5] Accellera Systems Initiative. Open Core Protocol specification, release 3.0, 2013.

[6] Sebastian Altmeyer, RobertI. Davis, and Claire Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority pre emptive systems. *Real-Time Systems*, 48(5):499–526, 2012.

[7] J. Bachrach, Huy Vo, B. Richards, Yunsup Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. Chisel: Constructing hardware in a scala embedded language. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1212–1221, June 2012.

[8] Ke Bai, A. Shrivastava, and S. Kudchadker. Stack data management for limited local memory (LLM) multi-core processors. In *Proc. of the International Conference on Application-Specific Systems, Architectures and Processors*, ASAP '11, pages 231–234. IEEE, 2011.

[9] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: an open toolbox for adaptive WCET analysis. In *Software Technologies for Embedded and Ubiquitous Systems - 8th IFIP WG 10.2 International Workshop, SEUS 2010, Waidhofen/Ybbs, Austria, October 13-15, 2010. Proceedings*, pages 35–46, 2010.

[10] M. Bekeman, A. Yoaz, F. Gabbay, S. Jourdan, M. Kalaev, and R. Ronen. Early load address resolution via register tracking. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 306–315, June 2000.

[11] Guillem Bernat and Niklas Holsti. Compiler support for WCET analysis: a wish list. In *Proceedings of the 3rd International Workshop on Worst-Case Execution Time Analysis, WCET 2003 - a Satellite Event to ECRTS 2003, Polytechnic Institute of Porto, Portugal, July 1, 2003*, pages 65–69, 2003.

[12] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. Cache-conscious data placement. *SIGPLAN Not.*, 33(11):139–149, October 1998.

[13] Hugues Cassé, Florian Birée, and Pascal Sainrat. Multi-architecture Value Analysis for Machine Code. In Claire Maiza, editor, *13th International Workshop on Worst-Case Execution Time Analysis*, volume 30 of *OpenAccess Series in Informatics (OASIcs)*, pages 42–52, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[14] K. D. Cooper and T. J. Harvey. Compiler-controlled memory. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems (ASPLOS-VIII), ACM SIGPLAN*, pages 2–11. ACM, 1998.

[15] Keith D. Cooper, Timothy J. Harvey, and Todd Waterman. Building a control-flow graph from scheduled assembly code. Technical Report TR02-399, Rice University, Houston, TX, USA 77005, 2002.

[16] Keith D. Cooper, Timothy J. Harvey, and Todd Waterman. Building a control-flow graph from scheduled assembly code. Technical report, 2002.

[17] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[18] Harvey G. Cragon. *Computer Architecture and Implementation*. Cambridge University Press, New York, NY, USA, 2000.

[19] M. de Michiel, A. Bonenfant, H. Casse, and P. Sainrat. Static loop bound analysis of c programs based on flow analysis and abstract interpretation. In *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA '08. 14th IEEE International Conference on*, pages 161–166, Aug 2008.

[20] J.-F. Deverge and I. Puaut. Wcet-directed dynamic scratchpad memory allocation of data. In *Real-Time Systems, 2007. ECRTS '07. 19th Euromicro Conference on*, pages 179–190, July 2007.

[21] Jean-Francois Deverge and Isabelle Puaut. Wcet-directed dynamic scratchpad memory allocation of data. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, ECRTS '07, pages 179–190, Washington, DC, USA, 2007. IEEE Computer Society.

[22] David R. Ditzel and H. R. McLellan. Register allocation for free: The c machine stack cache. *SIGPLAN Not.*, 17(4):48–56, March 1982.

[23] Angel Dominguez, Nghi Nguyen, and Rajeev K. Barua. Recursive function data allocation to scratch-pad memory. In *Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '07, pages 65–74, New York, NY, USA, 2007. ACM.

[24] DTU. D 2.1 software simulator of patmos. Technical report, T-CREST: `http://www.t-crest.org/page/results`, 2012.

[25] Tomasz Dudziak and Jörg Herter. Cache analysis in presence of pointer-based data structures. *SIGBED Rev.*, 8(3):7–10, September 2011.

[26] Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2-3):131–181, 1999.

[27] Stephen B. Furber. *VLSI RISC Architecture and Organization*. Marcel Dekker, Inc., New York, NY, USA, 1989.

[28] Jamie Garside and Neil C Audsley. Investigating shared memory tree prefetching within multimedia noc architectures. In *Memory Architecture and Organisation Workshop*, 2013.

[29] GNU. Extended Asm assembler instructions with c expression operands, 2015.

[30] Manil Dev Gomony, Benny Akesson, and Kees Goossens. Architecture and optimal configuration of a real-time multi-channel memory controller. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 1307–1312, 2013.

[31] Antonio González, Carlos Aliagas, and Mateo Valero. A data cache with multiple caching strategies tuned to different types of locality. In *ICS '95: Proceedings of the 9th international conference on Supercomputing*, pages 338–347, New York, NY, USA, 1995. ACM.

[32] Rodrigo González-Alberquilla, Fernando Castro, Luis Piñuel, and Francisco Tirado. Stack oriented data cache filtering. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '09, pages 257–266, New York, NY, USA, 2009. ACM.

[33] Daniel Grund. *Static Cache Analysis for Real-Time Systems: LRU, FIFO, PLRU.* PhD thesis, Saarland University, 2011.

[34] Daniel Grund, Jan Reineke, and Reinhard Wilhelm. A Template for Predictability Definitions with Supporting Evidence. In Philipp Lucas, Lothar Thiele, Benoit Triquet, Theo Ungerer, and Reinhard Wilhelm, editors, *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, volume 18 of *OpenAccess Series in Informatics (OASIcs)*, pages 22–31, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[35] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th annual Workshop on Workload Characterization*, 2001.

[36] Reinhold Heckmann and Christian Ferdinand. Worst-case execution time prediction by static program analysis. Technical report, AbsInt Angewandte Informatik GmbH. [Online, last accessed November 2013].

[37] M. Huang, J. Renau, S. M. Yoo, and J. Torrellas. L1 data cache decomposition for energy efficiency. In *Proceedings of the 2001 international symposium on Low power electronics and design (ISLPED '01), ACM SIGDA*, pages 10–15. ACM, 2001.

[38] Bach Khoa Huynh, Lei Ju, and A. Roychoudhury. Scope-aware data cache analysis for wcet estimation. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, pages 203–212, April 2011.

[39] Nicholas Jun Hao Ip and Stephen A. Edwards. A processor extension for cycle-accurate real-time software. In *Proceedings of the 2006 international conference on Embedded and Ubiquitous Computing*, EUC'06, pages 449–458, Berlin, Heidelberg, 2006. Springer-Verlag.

[40] Teresa L. Johnson and Wen-mei W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 315–326, New York, NY, USA, 1997. ACM.

[41] A. Jordan, F. Brandner, and M. Schoeberl. Static analysis of worst-case stack cache behavior. In *Proc. of the Conf. on Real-Time Networks and Systems*, pages 55–64. ACM, 2013.

[42] Sangyeol Kang and A.G. Dean. Leveraging both data cache and scratch-pad memory through synergetic data allocation. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*, pages 119–128, April 2012.

[43] A. Kannan, A. Shrivastava, A. Pabalkar, and Jong eun Lee. A software solution for dynamic stack management on scratch pad memory. In *Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific*, pages 612–617, 2009.

[44] E. Kasapaki and J. SparsØ. Argo: A time-elastic time-division-multiplexed noc using asynchronous routers. In *Asynchronous Circuits and Systems (ASYNC), 2014 20th IEEE International Symposium on*, pages 45–52, May 2014.

[45] Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Ingomar Wenzel. WCET Analysis: The Annotation Language Challenge. In Christine Rochange, editor, *7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)*, volume 6 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2007. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[46] Raimund Kirner and Peter Puschner. Time-predictable computing. In SangLyul Min, Robert Pettit, Peter Puschner, and Theo Ungerer, editors, *Software Technologies for Embedded and Ubiquitous Systems*, volume 6399 of *Lecture Notes in Computer Science*, pages 23–34. Springer Berlin Heidelberg, 2010.

[47] Marc Langenbach, Stephan Thesing, and Reinhold Heckmann. Pipeline modeling for timing analysis. In ManuelV. Hermenegildo and Germán Puebla, editors, *Static Analysis*, volume 2477 of *Lecture Notes in Computer Science*, pages 294–309. Springer Berlin Heidelberg, 2002.

[48] Hsien-Hsin S. Lee, Mikhail Smelyanskiy, Gary S. Tyson, and Chris J. Newburn. Stack value file: Custom microarchitecture for the stack. In *Proc. of the International Symposium on High-Performance Computer Architecture*, HPCA '01, pages 5–14. IEEE, 2001.

[49] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference*, DAC '95, pages 456–461, New York, NY, USA, 1995. ACM.

[50] Björn Lisper. Sweet – a tool for wcet flow analysis (extended abstract). In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*, volume 8803 of *Lecture Notes in Computer Science*, pages 482–485. Springer Berlin Heidelberg, 2014.

[51] Isaac Liu. *Precision Timed Machines*. PhD thesis, EECS Department, University of California, Berkeley, May 2012.

[52] Jing Lu, Ke Bai, and A. Shrivastava. Ssdm: Smart stack data management for software managed multicores (smms). In *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, pages 1–8, May 2013.

[53] T. Lundqvist and P. Stenstrom. A method to improve the estimated worst-case performance of data caching. In *Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth International Conference on*, pages 255–262, 1999.

[54] Nihar R. Mahapatra and Balakrishna Venkatrao. The processor-memory bottleneck: Problems and solutions. *Crossroads*, 5(3es), April 1999.

[55] V. Milutinovic, M. Tomasevic, B. Markovi, and M. Tremblay. A new cache architecture concept: the split temporal/spatial cache. In *Electrotechnical Conference, 1996. MELECON '96., 8th Mediterranean*, volume 2, pages 1108–1111 vol.2, May 1996.

[56] J. Mische, S. Uhrig, F. Kluge, and T. Ungerer. Using smt to hide context switch times of large real-time tasksets. In *Proc. of Intl. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 255–264, 2010.

[57] F. Mueller and D.B. Whalley. Fast instruction cache analysis via static cache simulation. In *Simulation Symposium, 1995., Proceedings of the 28th Annual*, pages 105–114, Apr 1995.

[58] Lena E. Olson, Yasuko Eckert, Srilatha Manne, and Mark D. Hill. Revisiting stack caches for energy efficiency. Technical Report TR1813, University of Wisconsin, Madison and AMD Research, December 2014.

[59] Soyoung Park, Hae-woo Park, and Soonhoi Ha. A novel technique to use scratch-pad memory for stack management. In *Proceedings of the*

Conference on Design, Automation and Test in Europe, DATE '07, pages 1478–1483, San Jose, CA, USA, 2007. EDA Consortium.

[60] Soyoung Park, Hae woo Park, and Soonhoi Ha. A novel technique to use scratch-pad memory for stack management. In *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, pages 1–6, April 2007.

[61] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design, Fourth Edition: The Hardware/Software Interface*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2008.

[62] Milos Prvulovic. The split spatial/non-spatial cache: A performance and complexity evaluation, 1999.

[63] P. Puschner and Ch. Koza. Calculating the maximum, execution time of real-time programs. *Real-Time Syst.*, 1(2):159–176, September 1989.

[64] Peter Puschner, Daniel Prokesch, Benedikt Huber, Jens Knoop, Stefan Hepp, and Gernot Gebhard. The T-CREST approach of compiler and WCET-analysis integration. In *9th Workshop on Software Technologies for Future Embedded and Ubiquitious Systems (SEUS 2013)*, pages 33–40, 2013.

[65] Arthur Pyka, Mathias Rohde, and Sascha Uhrig. A real-time capable first-level cache for multi-cores. In *In Workshop on High-performance and Real-time Embedded Systems HiRES 13*, Berlin, Germany, jan 2013.

[66] Rakesh Reddy and Peter Petrov. Eliminating inter-process cache interference through cache reconfigurability for real-time and low-power embedded multi-tasking systems. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '07, pages 198–207, New York, NY, USA, 2007. ACM.

[67] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. *ACM Trans. Embed. Comput. Syst.*, 4(4):751–778, November 2005.

[68] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *Proc. of the Conference on Hardware/Software Codesign and System Synthesis*, pages 99–108. ACM, 2011.

[69] J.A. Rivers and E.S. Davidson. Reducing conflicts in direct-mapped caches with a temporality-based design. In *Parallel Processing, 1996. Vol.3. Software., Proceedings of the 1996 International Conference on*, volume 1, pages 154–163 vol.1, Aug 1996.

[70] Jude A. Rivers, Edward S. Tam, Gary S. Tyson, Edward S. Davidson, and Matt Farrens. Utilizing reuse information in data cache management. In *Proceedings of the 12th International Conference on Supercomputing*, ICS '98, pages 449–456, New York, NY, USA, 1998. ACM.

[71] Christine Rochange, Pascal Sainrat, and Sascha Uhrig. *Time-Predictable Architectures*. Wiley-ISTE, 2014.

[72] S.Abbaspour and F. Brandner. Alignment of memory transfers of a time-predictable stack cache. In *Proc. of the 8th Junior Researcher Workshop on Real-Time Computing*, 2014.

[73] S.Abbaspour, A. Jordan, and F. Brandner. Lazy spilling for a time-predictable stack cache: Implementation and analysis. In *Proc. of the International Workshop on Worst-Case Execution Time Analysis*, volume 39 of *OASICS*, pages 83–92. Schloss Dagstuhl, 2014.

[74] Jesús Sánchez and Antonio González. A locality sensitive multi-module cache with explicit management. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, pages 51–59, New York, NY, USA, 1999. ACM.

[75] Marc Schlickling and Markus Pister. Semi-automatic derivation of timing models for wcet analysis. *SIGPLAN Not.*, 45(4):67–76, April 2010.

[76] Jörn Schneider and Christian Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, LCTES '99, pages 35–44, New York, NY, USA, 1999. ACM.

[77] M. Schoeberl. Time-predictable cache organization. In *Future Dependable Distributed Systems, 2009 Software Technologies for*, pages 11–16, March 2009.

[78] Martin Schoeberl. Jop: A java optimized processor. In *In Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2003*, pages 346–359. Springer, 2003.

[79] Martin Schoeberl. A time predictable instruction cache for a java processor. In Robert Meersman, Zahir Tari, and Angelo Corsaro, editors, *On the Move to Meaningful Internet Systems 2004: OTM 2004 Workshops*, volume 3292 of *Lecture Notes in Computer Science*, pages 371–382. Springer Berlin Heidelberg, 2004.

[80] Martin Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on*

Java Technologies for Real-Time and Embedded Systems (JTRES 2004), volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.

[81] Martin Schoeberl. Design and implementation of an efficient stack machine. In *Proceedings of the 12th IEEE Reconfigurable Architecture Workshop (RAW2005)*, Denver, Colorado, USA, April 2005. IEEE.

[82] Martin Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, vol. 2009, Article ID 758480:17 pages, 2009.

[83] Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015.

[84] Martin Schoeberl, Florian Brandner, Stefan Hepp, Wolfgang Puffitsch, and Daniel Prokesch. Patmos reference handbook, 2014.

[85] Martin Schoeberl, Florian Brandner, Jens Sparsø, and Evangelia Kasapaki. A statically scheduled time-division-multiplexed network-on-chip for real-time systems. In *Proceedings of the 6th International Symposium on Networks-on-Chip (NOCS)*, pages 152–160, Lyngby, Denmark, May 2012. IEEE.

[86] Martin Schoeberl, Benedikt Huber, and Wolfgang Puffitsch. Data cache organization for accurate timing analysis. *Real-Time Systems*, 49(1):1–28, 2013.

[87] Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W. Probst, Sven Karlsson, and Tommy Thorn. Towards a time-predictable dual-issue microprocessor: The Patmos approach. In *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, pages 11–20, Grenoble, France, March 2011.

[88] Vijayaraghavan Soundararajan and Anant Agarwal. Dribbling registers: A mechanism for reducing context switch latency in large-scale multiprocessors. Technical report, 1992.

[89] Lothar Thiele and Reinhard Wilhelm. Design for timing predictability. *Real-Time Syst.*, 28(2-3):157–177, November 2004.

[90] Eric Tune, Rakesh Kumar, Dean M. Tullsen, and Brad Calder. Balanced multithreading: Increasing throughput via a low cost multithreading hierarchy. In *Proc. of the 37th Intl. Symp. on Microarchitecture (MICRO 37)*, pages 183–194, Portland, Oregon, USA, 2004.

[91] Gary Tyson, Matthew Farrens, John Matthews, and Andrew R. Pleszkun. A modified approach to data cache management. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, MICRO 28, pages 93–103, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[92] S. Uhrig, S. Maier, and T. Ungerer. Toward a processor core for real-time capable autonomic systems. In *Signal Processing and Information Technology, 2005. Proceedings of the Fifth IEEE International Symposium on*, pages 19–22, Dec 2005.

[93] T. Ungerer, F.J. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quiñones, M. Gerdes, M. Paolieri, J. Wolf, H. Casse, S. Uhrig, I. Guliashvili, M. Houston, F. Kluge, S. Metzlaff, and J. Mische. Merasa: Multicore execution of hard real-time applications supporting analyzability. *Micro, IEEE*, 30(5):66–75, Sept 2010.

[94] Xavier Vera, Björn Lisper, and Jingling Xue. Data cache locking for higher program predictability. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '03, pages 272–282, New York, NY, USA, 2003. ACM.

[95] M.A. Viredaz and D.A. Wallach. Power evaluation of a handheld computer. *Micro, IEEE*, 23(1):66–74, Jan 2003.

[96] Lars Wehmeyer and Peter Marwedel. Influence of onchip scratchpad memories on wcet. In *Prediction, Proceedings of the 4th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2004.

[97] Randall T. White, Christopher A. Healy, David B. Whalley, Frank Mueller, and Marion G. Harmon. Timing analysis for data caches and set-associative caches. In *Proceedings of the Real-Time Technology and Applications Symposium*, RTAS '97, pages 192–203, 1997.

[98] Jack Whitham and Neil Audsley. Implementing time-predictable load and store operations. In *Proceedings of the seventh ACM international conference on Embedded software*, EMSOFT '09, pages 265–274, New York, NY, USA, 2009. ACM.

[99] Reinhard Wilhelm, Sebastian Altmeyer, Claire Burguière, Daniel Grund, Jörg Herter, Jan Reineke, Björn Wachter, and Stephan Wilhelm. Static timing analysis for hard real-time systems. In Gilles Barthe and Manuel

Hermenegildo, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 5944 of *Lecture Notes in Computer Science*, pages 3–22. Springer Berlin Heidelberg, 2010.

[100] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(7):966–978, 2009.

[101] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 30–44, New York, NY, USA, 1991. ACM.

[102] Jun Yan and Wei Zhang. Time-predictable multicore cache architectures. In *Computer Research and Development (ICCRD), 2011 3rd International Conference on*, volume 3, pages 1–5, March 2011.